Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Mejora de la Arquitectura de una Baliza Ultrasónica basada en un Dispositivo FPGA para un Sistema Local de Posicionamiento

Autor: Alejandro García Requejo

Tutor/es: Álvaro Hernández Alonso

2020

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Mejora de la Arquitectura de una Baliza Ultrasónica basada en un Dispositivo FPGA para un Sistema Local de Posicionamiento

Autor: Alejandro García Requejo

Tutor: Álvaro Hernández Alonso

Tribunal:

Presidente: Ignacio Bravo Muñoz

Vocal 1º: José Manuel Villadangos Carrizo

Vocal 2º: Álvaro Hernández Alonso

Calificación: .....

Fecha: .....

“El propósito de la educación es mostrar
a la gente cómo aprender por sí mismos.”
Noam Chomsky

AGRADECIMIENTOS

Este TFG es la culminación de todos los esfuerzos que he derramado en la idea que me rondó la cabeza hace años de llegar a ser ingeniero algún día.

En primer lugar, agradecer a mis padres y a mi hermana por enseñarme a pensar y aprender por mí mismo. A apoyarme en todo momento en cada una de mis decisiones y ayudarme estoicamente en todo lo posible y más, habiendo sido imposible llegar hasta aquí sin ellos.

A mis compañeros de carrera, en especial a Miguel y Talu por hacer de mi paso por la universidad un camino más fácil. A mis compañeros de prácticas en Indra por enseñarme tanto. Y a todos los amigos que Roma me hizo conocer formando parte de una experiencia Erasmus inolvidable.

A mis amigos de toda la vida que han hecho de estos años los mejores de mi vida repletos de experiencias imborrables.

A Virginia, que ha aguantado de la mejor forma posible mis jornadas intensas de estudio sin dejarme nunca de lado y mostrándome siempre su apoyo incondicional durante estos fantásticos años.

Y, por último, y no menos importante, a mi tutor, Álvaro Hernández, por su atención, cercanía, ayuda y motivación en todo momento para la realización de este TFG. Y por ejemplificar a la perfección lo que puede llegar a ser un gran profesor de universidad comprometido a las necesidades del alumnado.

Índice general

Índice de figuras

Índice de tablas

Parte I

Resumen

Resumen

Los sistemas de posicionamiento local o LPS se han convertido en una solución común para estimar la posición de un objeto o una persona en espacios interiores tales como una institución u organización, cubriendo las limitaciones de sistemas de posicionamiento más extendidos, como puede ser la tecnología GPS. A su vez el avance de dispositivos electrónicos de bajo coste se está viendo aumentado y mejorado en los últimos años, permitiendo una reducción de las infraestructuras de algunos proyectos. Por todo esto, el interés de este Trabajo Fin de Grado se centra en mejorar la arquitectura de un sistema LPS basado en tecnología ultrasónica, también conocido como ULPS, para que éste tenga integrado en un dispositivo FPGA, en el que se desarrolla la aplicación, un periférico de bajo coste que consiga una comunicación WiFi con otros elementos externos, como un PC.

Palabras clave: LPS (*Local Positioning System*), FPGA (*Field Programmable Gate Array*), SoC (*System-on-Chip*).

Abstract

Local positioning systems or LPS have become a common solution to estimate the position of an object or a person in indoor spaces, such as institutions or organizations, covering the limitations of more widespread positioning systems, such as GPS technology. At the same time, the advancement of low-cost electronic devices has been increasing and improving in recent years, allowing a reduction in the infrastructure of some projects. For all this, the interest of this Final Degree Thesis focuses on improving the architecture of an LPS system based on ultrasonic technology, also known as ULPS, so that a FPGA device, in which the application is developed, integrates a low-cost peripheral that achieves WiFi communication with other external elements, such as a PC.

Keywords: LPS (*Local Positioning System*), FPGA (*Field Programmable Gate Array*), SoC (*System-on-Chip*).

Resumen Extendido

En la actualidad, existen numerosas tecnologías que determinan la posición de un móvil que se desplaza por un área, siendo la tecnología GPS la más extendida. Concretamente, la tecnología GPS localiza un móvil con exactitud en extensas áreas localizadas en el exterior. Aun así, esta tecnología no cubre las necesidades de localización que puede requerir una institución para localizar un objeto o una persona en un espacio cerrado. Para esto último, se han desarrollado numerosos sistemas, bajo el nombre de LPS (*Local Positioning System*), que con distintas tecnologías tratan de resolver dicha necesidad de localización.

Al contrario que con la tecnología GPS, la cual es la tecnología predominante en sistemas de localización exteriores, los sistemas LPS no poseen una tecnología que destaque por encima de otra, siendo todas utilizadas para determinadas situaciones. Aunque existen sistemas LPS basados en radio frecuencia, en infrarrojos, en luz visible, etc., en este trabajo se atenderá a un sistema basado en ultrasonidos, conociéndose como sistema ULPS (*Ultrasonic Local Positioning System*).

Un sistema ULPS, como cualquier sistema de posicionamiento, está formado por un emisor o baliza, que emitirá las señales ultrasónicas a través de un transductor ultrasónico, y un receptor, que calculará su posición en función del tiempo de llegada de las señales recibidas por parte del emisor. En el presente Trabajo Fin de Grado, el estudio y desarrollo se centrará en la parte del emisor, el cual está basada en un dispositivo FPGA. Concretamente, en un SoC (*System on Chip*) que incorpora un procesador, junto con la propia FPGA.

Un sistema ULPS al completo posee varios emisores confinados en una única plataforma, llamada baliza, que a su vez están repetidos por toda el área de extensión interior de la que se desea tener localización, concretamente en el techo. Estas balizas deben ser fácilmente configurables, ya que dependiendo del lugar donde estén colocadas dentro del espacio interior, será útil usar un esquema de emisión u otro. Las modulaciones consisten en variar algún parámetro de la onda ultrasónica a transmitir y dependiendo del entorno pueden sufrir más o menos variaciones o interferencias que pueden llegar erróneamente al receptor.

Para que estos sistemas sean configurables de una manera sencilla es útil que el operador que esté controlando el sistema pueda configurarlo rápidamente a

través de su ordenador y de forma inalámbrica. Para ello es necesario que las balizas tengan incorporadas, aprovechando el avance de dispositivos de bajo coste, un módulo WiFi al que poder enviar la información de configuración de las balizas y que el propio sistema sea capaz de llevarlo a cabo. Esta aplicación comentada es la que se desarrollará y posteriormente se explicará en el actual documento. Para llevarlo a cabo, se ha escogido un periférico WiFi comercial que ha sido incorporado al SoC integrado en el emisor del sistema ULPS.

El sistema ULPS tomado como punto de partida ha sido el realizado por el grupo GEINTRA en el trabajo "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*" [1]. En este trabajo, en primer lugar, se han eliminado los módulos y la programación software que poseía el sistema frente a una conexión ethernet previa, para después proceder a la integración del periférico.

La integración del periférico ha dado lugar a distintas fases de desarrollo para que el SoC utilizado fuese capaz de comunicarse con el módulo WiFi perfectamente. La primera etapa de desarrollo ha consistido en el desarrollo de un módulo hardware específico dentro de la FPGA del SoC, para que se comunicase con el periférico WiFi sin necesidad de que el procesador que incorpora el SoC deba preocuparse de ello. Y, por último, ha sido necesario un desarrollo dentro del propio controlador para que a través de una pila de protocolos TCP/IP se puedan transmitir y recibir los mensajes de una forma correcta con un ordenador exterior al sistema ejecutando un software específico, conectados, tanto el ordenador como el sistema ULPS, a un mismo router WiFi.

Parte II

Memoria

Capítulo 1

Introducción

La tecnología GPS es, actualmente, la tecnología dominante en espacios exteriores. Aun así, carece de gran precisión en espacios interiores o en ciertas circunstancias con visibilidad reducida. Para lidiar con este problema se ha empezado a investigar sobre diferentes tecnologías que permitan, en espacios interiores, poseer dicha precisión, aunque no se ha llegado a ninguna solución definitiva. Algunas de estas tecnologías involucradas en estos sistemas, llamados LPS (*Local Positioning Systems*), se han basado en radio frecuencia, infrarrojos, ultrasonidos o luz visible.

En este Trabajo de Fin de Grado se abordará en concreto un sistema ULPS (*Ultrasonic LPS*) previamente desarrollado para, posteriormente, realizar una mejora en su arquitectura. Comúnmente, un sistema ULPS está formado por múltiples trasmisores ultrasónicos o balizas distribuidos por todo el espacio interior donde se quiere tener cobertura, de esta manera, múltiples receptores que se muevan por el área de alcance pueden detectar las transmisiones y determinar los tiempos de llegada, llamado TDOA (*Time Difference Of Arrival*), que junto a un algoritmo de posicionamiento sea capaz de estimar su posición.

Cabe destacar, que los espacios reales poseen condiciones complejas para los sistemas LPS que hacen difícil su operación. Las señales pueden verse afectadas por ruido e interferencias que influyan erróneamente en la determinación del TDOA y por tanto en la determinación de la posición. Para solventar este problema se puede hacer uso de codificaciones y modulaciones en las señales transmitidas que permitan una mejor obtención del TDOA ante ambientes ruidosos.

La codificación en las transmisiones ultrasónicas, además, implica un incremento computacional importante que necesita de procesadores muy potentes en los sistemas. Los dispositivos FPGA, ante esto, son una gran opción para este tipo de aplicaciones que necesitan procesar una gran cantidad de datos, proporcionando paralelismo a las operaciones que han de realizarse, además de la flexibilidad que ya poseen los procesadores. Por ello, este proyecto se ha llevado a cabo con un SoC, que incluye en el dispositivo FPGA un sistema de

procesamiento ya incorporado, del que se pueden aprovechar las ventajas, tanto, de la FPGA, como, las del procesador.

Otro elemento importante pasa por que el sistema desarrollado sea perfectamente configurable de forma rápida y eficaz para poder explorar distintas modulaciones o codificaciones en los distintos medios donde se instale. Para ello, será necesario una comunicación entre un PC y el propio sistema a configurar en el que los elementos desplegados para ello sean de un coste y tamaño reducido. En este proyecto, aprovechando que los dispositivos WiFi de bajo coste, están viéndose mejorados en los últimos años, se desarrollará una aplicación de comunicación entre el sistema y un PC externo, para que se pueda configurar correctamente.

1.1. Contexto

Este proyecto se enmarca en el grupo de investigación GEINTRA, *Grupo de Ingeniería Electrónica Aplicada a Espacios Inteligentes y Transporte*, perteneciente al Departamento de Electrónica de la Escuela Politécnica Superior de la Universidad de Alcalá.

El grupo de investigación GEINTRA posee una gran experiencia en sistemas de posicionamiento. En concreto, este trabajo tiene como referencia el proyecto TARSIUS realizado por el grupo, el cual trata de investigar el desarrollo de tecnologías de localización que puedan suplir a los sistemas GNSS (*Global Navigation Satellite System*) en lugares donde éstos no pueden desarrollarse. Para ello existen distintos niveles de desarrollo dentro del proyecto, desde el desarrollo a más bajo nivel, basado en la implementación de arquitecturas basadas en FPGAs, como el desarrollo a más alto nivel de las aplicaciones móviles que permiten visualizar al objeto localizado por el sistema de posicionamiento [2].

1.2. Objetivos y motivación

El objetivo principal de este Trabajo de Fin de Grado será mejorar la arquitectura presente de una baliza ultrasónica basada en un dispositivo FPGA: en concreto, transformar el medio físico bajo el que se rige la comunicación de la baliza ultrasónica con un PC externo encargado de configurarla. Este cambio consiste en sustituir la comunicación ethernet entre el sistema y un router WiFi, que permitía conexión inalámbrica, e incorporar un periférico/adaptador WiFi de bajo coste conectado al sistema, haciendo que éste se conecte directamente de forma

inalámbrica con un router que esté conectado a su vez con el PC externo que configura el sistema.

Para llegar a cumplir este objetivo principal es necesario dividirlo en objetivos más específicos:

- Análisis de sistemas LPS previos, haciendo énfasis en sistemas ULPS.
- Análisis de los dispositivos en los que se pueden implementar los sistemas LPS, en concreto procesadores, FPGAs y SoCs.
- Análisis de los distintos dispositivos que formarán la arquitectura final del sistema.
- Estudio del sistema previo en el que se basa el proyecto.
- Estudio del periférico WiFi que se desea incorporar en el sistema.
- Implementación de la comunicación entre el dispositivo FPGA y el periférico para que éste pueda ser utilizado correctamente bajo una pila TCP/IP desarrollada específicamente para dicho dispositivo.
- Implementación de la aplicación Cliente-Servidor para que el dispositivo FPGA se conecte al PC externo a través de la una conexión WiFi.
- Validación de las propuestas y aportaciones realizadas.

1.3. Estructura del Documento

La redacción del presente Trabajo Fin de Grado se ha dividido en siete capítulos que atienden a la siguiente explicación asociada:

- **Capítulo 1:** se presenta el Trabajo Fin de Grado atendiendo al contexto, los objetivos del proyecto y la estructura del documento.
- **Capítulo 2**: se explican todos los temas necesarios para el entendimiento del resto del documento, además de explicar sistemas previos relacionados con el presente proyecto.
- **Capítulo 3**: se aborda la arquitectura del sistema, explicando todos los elementos que formarán el sistema final.

- **Capítulo 4**: se estudia en profundidad el sistema previo que se toma de partida y sobre el que se realizarán las mejoras abordadas en este Trabajo Fin de Grado.

- **Capítulo 5**: se detalla cómo se han desarrollado las mejoras del sistema en todas sus etapas, con ayudas de flujogramas, máquinas de estado y esquemas.

- **Capítulo 6**: se muestran los resultados realizados sobre el sistema final ya implementado, así como los recursos utilizados por el dispositivo FPGA.

- **Capítulo 7**: se muestran las conclusiones y los trabajos futuros que pueden mejorar el sistema.

Capítulo 2

Antecedentes

Para llegar a comprender el proyecto que aborda este Trabajo Fin de Grado es necesario exponer los fundamentos teóricos en los que se basa. Estos conocimientos previos serán explicados en este capítulo, tratando, en primer lugar, los diferentes sistemas de posicionamiento y las tecnologías en las que se basan, para terminar, abordando los dispositivos utilizados para llevar a cabo su desarrollo que permiten el procesamiento digital de señal necesario en estas aplicaciones.

2.1. Sistemas de Posicionamiento

Un sistema de posicionamiento trata de posicionar un elemento en un entorno conocido, pudiendo ubicarlo, en primer lugar, y seguir su desplazamiento, posteriormente. Existen distintos tipos de sistemas dependiendo de si el elemento que se desea ubicar se encuentra dentro de una gran área exterior como puede ser una ciudad, de la que se quiere saber la calle en la que está. O, si, por el contrario, el elemento se encuentra en una pequeña área interior, como un hospital y se desea establecer con precisión su ubicación dentro del edificio. De esta manera se pueden distinguir, respectivamente, los sistemas de posicionamiento en exteriores o global y los sistemas de posicionamiento en interiores o local.

2.1.1. Sistemas de Posicionamiento en Exteriores

Los sistemas de posicionamiento en exteriores son sistemas globales que permiten estimar la ubicación de un móvil en un mapa conocido ubicado en el exterior. El sistema de posicionamiento en exteriores más extendido es el GPS (*Global Positioning System*), que se explicará a continuación. Otros sistemas globales de navegación por satélite o GNSS (*Global Navigation Satellite System*)

han sido desarrollados recientemente como GLONASS, utilizado por el Ministerio de Defensa de la Federación Rusa, o Galileo, desarrollado por la Unión Europea.

2.1.1.1. GPS

El Sistema de Posicionamiento Global o GPS es un sistema desarrollado por el Departamento de Defensa de Estados Unidos que permite determinar la ubicación de un objeto en toda la Tierra con la hora exacta de su localización [3]. Para ello el GPS dispone de satélites en órbita alrededor de la Tierra, como se observa en la figura 2.1, de estaciones terrestres de seguimiento y control, y receptores GPS que pertenecen a los usuarios. Los satélites transmiten señales que una vez son recibidas por los receptores, éstos son capaces de calcular por separado las coordenadas de altitud, longitud y latitud, junto con la hora local precisa.

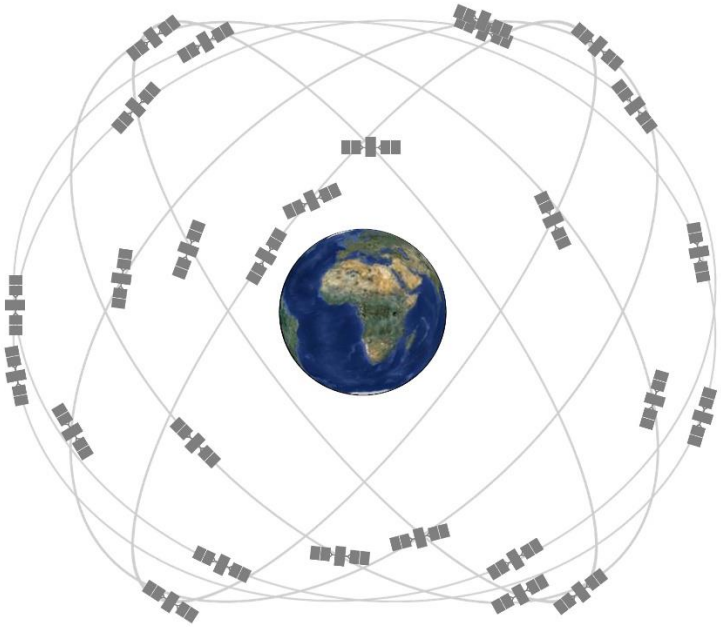


Figura 2.1. Constelación de los satélites GPS alrededor de la Tierra[1].

2.1.1. Sistemas de Posicionamiento en Interiores

Los sistemas de posicionamiento en interiores son sistemas de posicionamiento local o LPS *(Local Positioning System*) que permiten obtener la localización de un objeto que está dentro de un espacio cerrado, como puede ser un hospital, una universidad, una casa, etc. De esta manera, estos sistemas de

[1] https://www.gps.gov/multimedia/images/constellation.jpg

posicionamiento permiten cubrir las limitaciones del sistema GPS, que es incapaz de ubicar un objeto en un espacio interior y con menor precisión. Para realizar estos sistemas es necesario la colocación, dentro del propio espacio, de balizas que haciendo uso de diferentes tecnologías, que se explicarán a continuación, permiten obtener la distancia a ellas mediante la temporización del tiempo de vuelo, medida de fase, etc. Al contrario que en los sistemas de posicionamiento global, no existe una tecnología predominante que satisfaga los sistemas de posicionamiento en interiores, resultando de ello un gran número de tecnologías que ofrecen ventajas e inconvenientes para estos sistemas.

2.1.1.1 Infrarrojos

Es un tipo de radiación electromagnética cuya longitud de onda es más larga que la de la luz visible, siendo, de este modo, invisible para el ojo humano. Uno de los métodos más utilizado en la explotación de la tecnología infrarroja para sistemas de posicionamiento consiste en el despliegue de puntos de accesos distribuidos por cada una de las habitaciones de una institución, actuando como receptores, y emisores que emiten identificadores globales únicos (*Globally Unique Identifier o GID*) cada 15 segundos.

2.1.1.2 Radio Frecuencia

Ésta es una de las tecnologías más usadas en el posicionamiento local, ya que permite cubrir grandes distancias entre emisor y receptor, cruzando las paredes del edificio, siendo menos invasiva la instalación requerida. Además, como puede ocurrir con los sistemas basados en WiFi, no se requiere la instalación de nuevos sistemas.

2.1.1.3 Visión

La localización basada en visión hace uso de sensores ópticos para adquirir la información necesaria para la localización. Por otro lado, es necesario una información detallada del lugar donde tiene lugar la localización y visión directa del móvil a posicionar.

2.1.1.4 Ultrasonidos

Los sistemas de posicionamiento basados en ultrasonidos se basan en la emisión y recepción de ultrasonidos en la que el receptor calcula la diferencia de tiempo de llegada o TDOA (*Time Difference of Arrival*) y, junto un algoritmo de

posicionamiento, consiguen determinar la localización del objeto. En la figura 2.2 se puede observar un ejemplo de un sistema basado en ultrasonidos, pudiéndose observar los emisores, que consisten en balizas repartidas por el techo del área en el que se lleva a cabo el posicionamiento, emitiendo señales ultrasónicas.

Figura 2.2. Vista global de un laboratorio donde está instalado una baliza[2].

2.2. Dispositivos para el Procesamiento Digital de Señal

Los sistemas de posicionamiento comentados anteriormente deben implementar funciones matemáticas de complejidad variable, bien para desarrollar los algoritmos de posicionamiento o bien, para generar las señales a transmitir que hacen uso de modulaciones digitales. Este tipo de operaciones matemáticas necesarias para el llamado procesamiento digital de señal, necesita de dispositivos especialmente seleccionados para realizar dicho cómputo, en los que se pueden distinguir dispositivos programables o configurables.

2.2.1. Sistemas Programables

Un sistema programable es un circuito electrónico que contiene un microprocesador integrado, permitiendo así realizar múltiples funciones

[2] Imagen extraída de: Hernández A, Garcia E, Gualda D, Villadangos J.M., Nombela F, Ureñá J, "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*", IEEE transactions on instrumentation and measurement, vol. 66, no. 8, August 2017, pp. 1954-1964.

mediante un programa informático o software almacenado en una memoria interna. El microprocesador es el núcleo del sistema y puede ser remplazado por un microcontrolador o un procesador digital de señal (*Digital Signal Processor, DSP*) dependiendo de la aplicación específica. En este caso se estudiarán los posibles dispositivos programables a utilizar para el tratamiento digital de señales.

2.2.1.1. Microcontroladores

Un microcontrolador es un circuito integrado, que además de una unidad de procesamiento o CPU, como puede incorporar un microprocesador genérico, contiene elementos de memoria y periféricos de entrada y salida, como se puede observar en la figura 2.3.

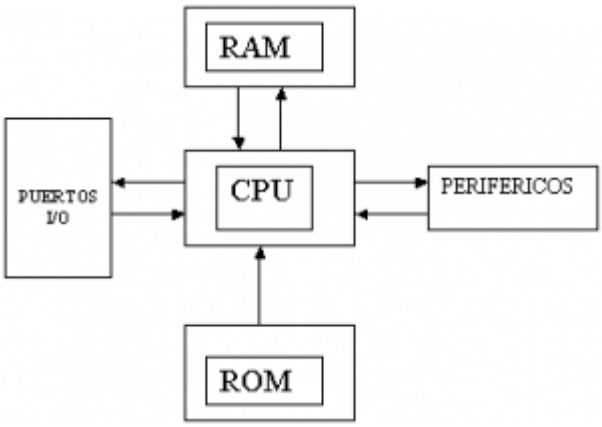


Figura 2.3. Elementos que componen un microcontrolador[3].

A la hora de realizar tareas de procesamiento digital, un microcontrolador puede verse limitado al realizar dichas operaciones las cuales requieren de una gran carga computacional. Sin embargo, es cierto que en algunas aplicaciones donde no haya una carga suficientemente grande de operaciones de procesamiento digital, un microcontrolador bien elegido puede ser suficiente.

2.2.1.2 Procesador Digital de Señales

Un procesador digital de señales o DSP (*Digital Signal Processor*) es un microprocesador con una arquitectura específicamente diseñada para acelerar los cálculos matemáticos intensos implicados en la mayoría de los sistemas de procesado digital de señal [4]. Para ello, la CPU de los DSPs disponen de

[3] https://www.electronicaestudio.com/wp-content/uploads/2018/10/micro_esq1-300x210.png

unidades computacionales específicas que pueden trabajar en paralelo, en la que destacan las unidades MAC, que ejecutan de forma rápida operaciones de multiplicar y acumular. Otra característica importante, es que dispone de unidades generadoras de direcciones de datos que realizan los cálculos de la nueva dirección necesaria para el acceso a los operandos. También incluyen arquitecturas de memoria que permiten un acceso múltiple para permitir de forma simultánea cargar varios operandos. Por último, los DSPs incluyen periféricos e interfaces de entrada y salida integrados en el chip que permiten que el procesador se comunique con ellos rápidamente al igual que un microcontrolador. En la figura 2.4, se puede observar la arquitectura típica de un DSP.

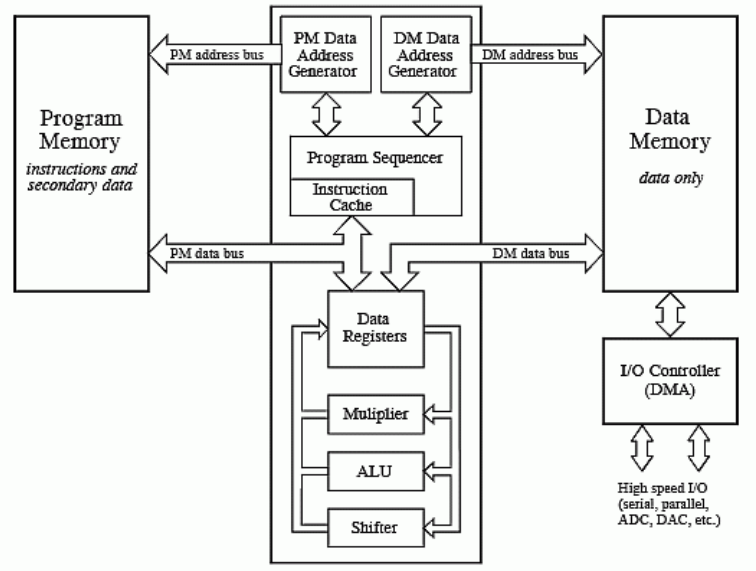


Figura 2.4. Arquitectura típica de un DSP[4].

Las características vistas del DSP hacen de él un procesador ideal para llevar a cabo tareas de procesamiento de señal de altas prestaciones que son numéricamente intensas y repetitivas. Esto hace que su elección para este tipo de aplicaciones sea mejor que un microcontrolador que se verá afectado por una gran carga computacional.

2.2.2. Sistemas Configurables

Los sistemas configurables son sistemas electrónicos digitales cuya función se puede modificar utilizando elementos que lo componen y cambiando la interconexión entre ellos. Esta funcionalidad permite crear arquitecturas hardware específicas para cualquier aplicación, a diferencia de los sistemas

[4] https://www.dspguide.com/graphics/F_28_5.gif

programables, que con una misma arquitectura hardware modifican el funcionamiento lógico del sistema a través de un programa o software.

Esta característica hace que los sistemas configurables sean una importante solución a los sistemas de tratamiento digital de señal debido a que pueden destinar hardware específico para realizar dichas operaciones. De esta manera, se pueden realizar en un solo ciclo de reloj operaciones que en un DSP puede requerir de cientos de ciclos de reloj. A continuación, se describirán los sistemas configurables más utilizados.

2.2.2.1. ASIC

Un circuito integrado de aplicación específica o ASIC (*Application-Specific Integrated Circuit*) es, como su nombre indica, un dispositivo creado para una aplicación específica [5]. Estos chips son realizados completamente en fábrica para su aplicación específica que generalmente se llevan a cabo para un producto que tendrá una gran producción y de esta manera, el ASIC pueda contener una gran parte de la electrónica del sistema completo. Además, pueden contener funciones analógicas y digitales. Y, aunque su desarrollo es costoso, lento y requiere muchos recursos, los ASICs ofrecen un rendimiento muy alto junto con un bajo consumo de energía. En la figura 2.5 se pueden observar los tipos de dispositivos ASIC que se pueden encontrar, aunque normalmente, los dispositivos ASIC suele estar asociar a las formas de diseño Full-Custom y Semi-Custom, que se explican a continuación.

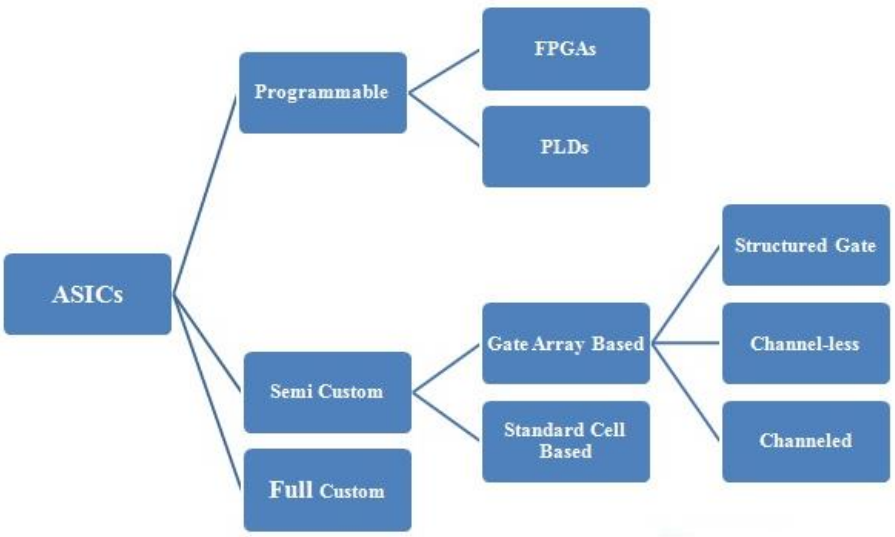


Figura 2.5. Tipos de dispositivos ASICs[5].

[5] https://www.elprocus.com/wp-content/uploads/Types-of-ASICss.jpg

- **Full-Custom**: en este tipo, el diseño está realizado totalmente a medida, siendo predefinidas todas las capas del dispositivo. Estos dispositivos permiten un área reducida y una mayor integración de componentes analógicos al sistema, aunque requieren de una elevada complejidad y un elevado coste de diseño.

- **Semi-Custom**: estos sistemas están realizados a través de bloques ya existentes, a priori no conectados entre sí. Para conseguir la funcionalidad final del dispositivo se deben conectar apropiadamente. Con esto se reducen los costes y la dificultad del diseño. Según el tipo de celdas lógicas existentes y la cantidad de personalización en las interconexiones, estos ASIC se dividen en ASICs basados en células estándar (*Standards-Cells*) y ASICs basados en matriz de puertas (*Gate-Arrays*).

Por otro lado, desmarcándose de los dispositivos ASIC, existen dispositivos de lógica programable, cuya funcionalidad puede ser cambiada al momento por un programador que describa el hardware que tendrá el dispositivo mediante una aplicación informática. Dentro de estos dispositivos existen los PLDs y las FPGAs. En nuestros días, es mayor el uso de las FPGAs debido a la gran flexibilidad que ofrece su diseño frente a los PLDs.

2.2.2.2. FPGA

Las FPGAs (*Field Programmable Gate Arrays*) son dispositivos electrónicos reconfigurables cuya arquitectura, mostrada en la figura 2.6, está basada en matrices de bloques lógicos configurables o CLBs (*Configurable Logic Blocks*) conectados entre sí y entre los bloques de entrada salida o IOBs (*Input Output Blocks*) mediante interconexiones programables. La interconexión se consigue mediante el volcado de un mapa de bits del diseño en una memoria de la propia FPGA que genera un programa informático a partir de la descripción hardware realizada mediante un lenguaje destinado a ello, como puede ser el lenguaje VHDL o Verilog.

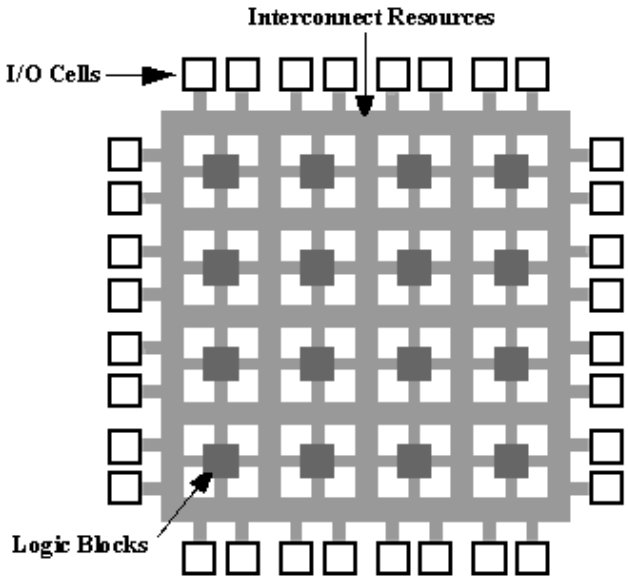


Figura 2.6. Arquitectura interna de una FPGA[6]

Las FPGAs se han convertido en dispositivos electrónicos muy demandados debido a la capacidad de reprogramación y flexibilidad que presenta frente a otros dispositivos configurables como los PLDs o los ASICs. Y también frente a los dispositivos programables como microcontroladores o DSPs debido a su capacidad de procesamiento en paralelo que ofrece el diseño hardware. Esto hace, que las FPGAs sean dispositivos perfectos para aplicaciones en la que se deba desarrollar algoritmos de tratamiento digital de señal.

2.2.2.3. SoC

Un SoC (*System on Chip*) es un circuito integrado que contiene uno o varios procesadores en su interior, como microprocesadores, microcontroladores y/o procesadores digitales de señal (DSPs), aparte de todos los dispositivos electrónicos que se puedan añadir, como módulos de memoria, osciladores, PLLs (*Phase Locked Loops*), DACs (*Digital-to-Analogue Converters*, ADCs (*Analog-to-Digital Converters)*, etc. En la figura 2.7 se puede ver un ejemplo de los elementos que puede albergar, junto con la CPU, un SoC destinado a *smartphones*.

[6] https://upload.wikimedia.org/wikipedia/commons/7/7d/Fpga1a.gif

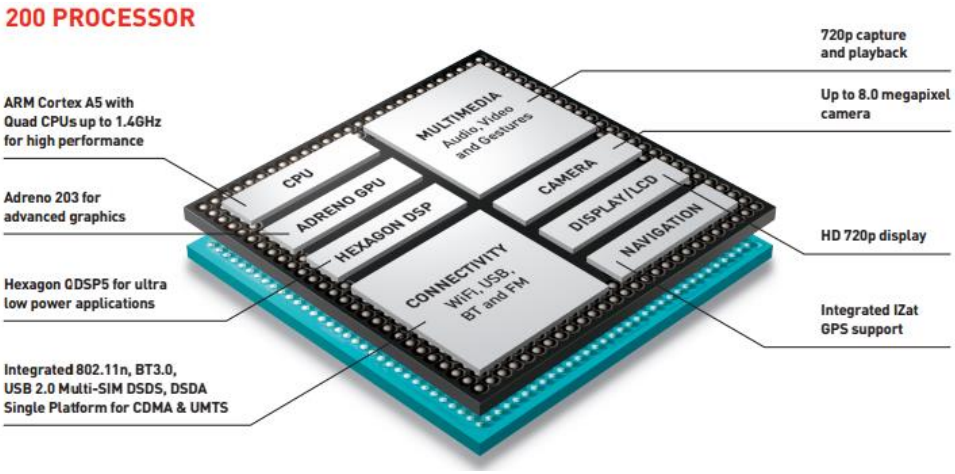


Figura 2.7. Arquitectura de un System on Chip destinado a smartphones[7].

Existe una tendencia cada vez más frecuente de integrar en un SoC, además de un procesador, una FPGA. Este tipo de SoCs permite albergar en un solo circuito ambos elementos, pudiendo hacer uso de las ventajas que pueden ofrecer cada uno de ellos. Un claro ejemplo de esta fabricación son los dispositivos Zynq desarrollados por el fabricante XILINX, que integra un sistema completo basado en procesador ARM junto con una FPGA.

[7] https://i.pinimg.com/originals/87/8c/38/878c389f3c426aaa20813e9709c5d68c.png

Capítulo 3

Arquitectura Propuesta

El presente Trabajo de Fin de Grado se centra en la incorporación de un módulo WiFi a un sistema ULPS, para que éste pueda ser configurado de forma inalámbrica sin necesidad de una conexión física basada en Ethernet, tal y como se presentaba el sistema previamente. La arquitectura completa del sistema constará de un software externo al sistema encargado de configurar los parámetros del sistema ULPS, un módulo WiFi cuya función es la de recibir dicha información de forma inalámbrica y configurar el sistema, un router que actúe como un punto de acceso en la red inalámbrica, una tarjeta Zedboard encargada de generar las señales moduladas y realizar la interconexión física de los componentes y los módulos encargados de interactuar con los transductores que forman el sistema ULPS. Dicha arquitectura se muestra en la figura 3.1 en la que se pueden identificar todos los módulos que conforman el sistema.

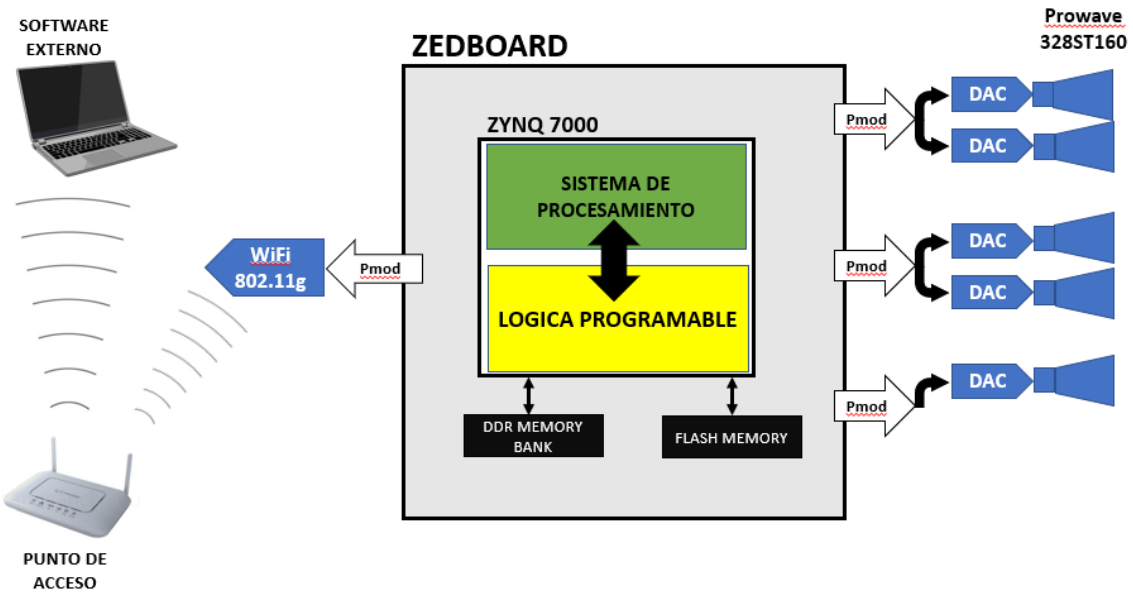


Figura 3.1. Arquitectura del sistema completo.

3.1. Módulo Hardware Externo

Para la configuración del sistema se han hecho modificaciones sobre un software desarrollado en Matlab, para que éste, siendo ejecutado desde un PC con conexión WiFi, se comunique con el módulo WiFi insertado en la tarjeta Zedboard y de esta manera, configure el sistema ULPS.

El software consiste en la creación de un cliente TCP para que se conecte al módulo WiFi del correspondiente sistema ULPS que en su caso actúa como servidor TCP.

Mediante esta comunicación se envían los códigos de una señal modulada previamente guardada en un fichero, junto con el periodo de transmisión, la activación del sistema y la habilitación del maestro, en el caso de que se quiera activar.

Los elementos modificables en la interfaz gráfica mostrada en la figura 3.2, son los siguientes:

- Periodo de transmisión.
- Ficheros de datos que contienen los códigos de las señales a ser enviadas. Dichos ficheros contienen las señales digitales que la Zedboard reproducirá en valores de 0 a 4095 digitalmente.
- IP y puerto de conexión del LPS que quiere configurarse.
- Habilitación del sistema como esclavo o maestro. Si está habilitado como maestro, todas las balizas del sistema transmitirán las mismas señales.
- Habilitación del sistema.

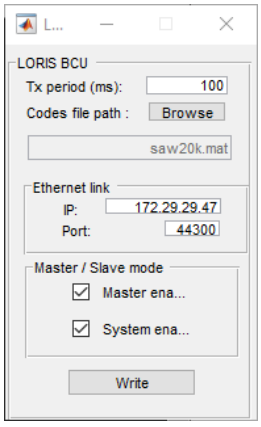


Figura 3.2. Interfaz del software externo.

3.2. Módulo WiFi

Se ha utilizado un módulo WiFi de la familia de periféricos Pmod ofrecidos por Digilent debido a la fácil conexión con la tarjeta Zedboard [6]. En concreto, se ha escogido el módulo WiFi Interface 802.11g, mostrado en la figura 3.3, el cual está basado en el módulo RF de Microchip MRF24WG0MA.

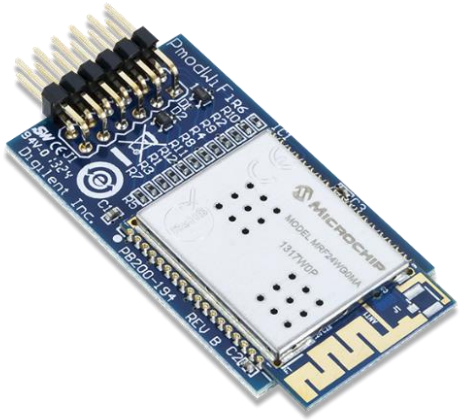


Figura 3.3. Pmod WiFi Interface 802.11g[8].

El chip MRF24WG0MA proporciona una conexión WiFi compatible con el estándar de modulación 802.11g, la cual utiliza la banda de 2,4 GHz. Aunque el chip tiene una antena PCB integrada, permite la conexión de una antena exterior a través de un conector coaxial en miniatura, como es el caso del Pmod WiFi.

El diagrama de bloques del módulo se puede observar en la figura 3.4. La comunicación con el módulo se realiza a través del protocolo de comunicación SPI, concebido para la transferencia entre circuitos integrados de dispositivos electrónicos mediante una comunicación de 4 hilos, además de las señales de interrupción, hibernar, reset, la señal trace para depuración y las señales de alimentación [14].

[8] Imagen extraída de: https://store.digilentinc.com/pmodwifi-wifi-interface-802-11g/

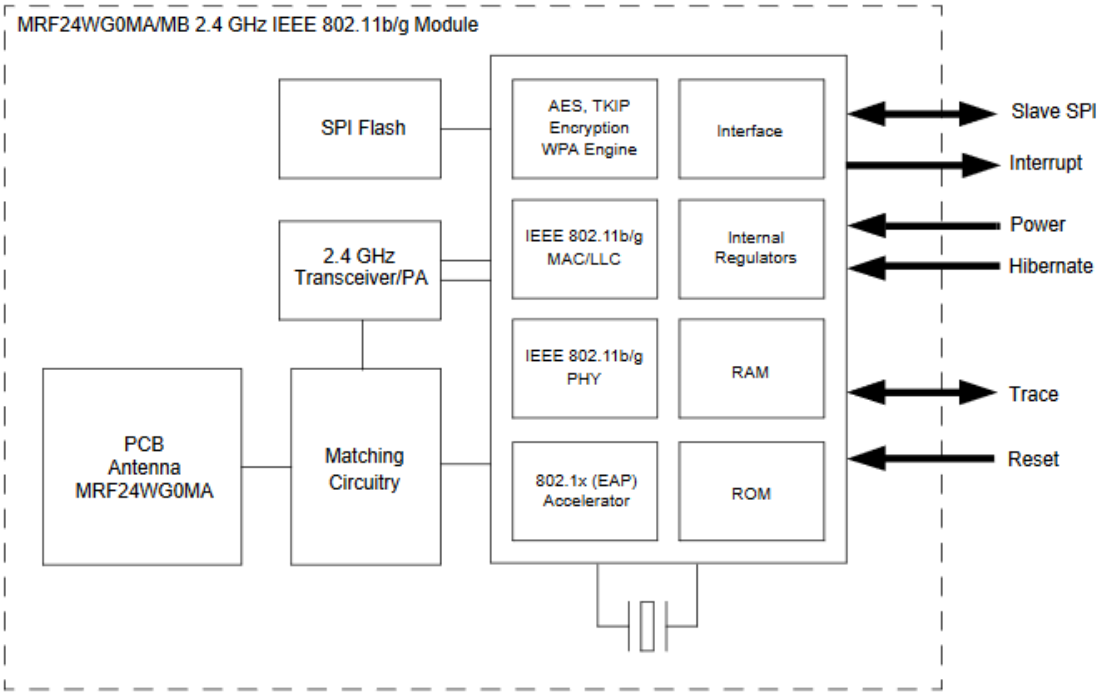


Figura 3.4. Diagrama de bloques del módulo MRF24WG0MA[9].

Está diseñado para ser utilizado a través de un microcontrolador PIC de la familia de Microchip a través de la pila TCP/IP diseñada por la misma empresa. Por lo tanto, ha sido necesario escoger una pila TCP/IP modificada por Digilent en base a la creada por Microchip, denominada deIP (*Digilent Embbedded IP STACK*) [15]. Esta modificación, disponible en código abierto, permite, a partir de modificaciones especificas a cada arquitectura, ser compilada en cualquier entorno de desarrollo y ejecutada en un número mayor de microcontroladores.

3.3. Punto de Acceso

Para que haya comunicación WiFi entre el módulo y el ordenador que implemente el software en Matlab, es necesario disponer de un router conectado a la misma red que funcione como un punto de acceso inalámbrico. De este modo, dichos componentes formarán una subred inalámbrica en la que tanto el ordenador, como el módulo WiFi, deberán estar conectados al punto de acceso para tener comunicación entre ellos, sin necesidad de ningún otro elemento de configuración en dichos dispositivos.

[9] Imagen extraída de: http://ww1.microchip.com/downloads/en/DeviceDoc/70686B.pdf

El punto de acceso utilizado es un router comercial, el cual debe ser capaz de recibir señales WiFi con referencia a la IP de un dispositivo y retransmitirlas a la dirección IP del dispositivo destino, todos ellos dentro de la misma subred.

3.4. Plataforma de Desarrollo Zedboard

La Zedboard (*Zynq Evaluation & Development Board*), mostrada en la figura 3.5, es una tarjeta de desarrollo basada en el dispositivo Zynq-7000 SoC de Xilinx. Además, incorpora múltiples conexiones para facilitar la comunicación con otros dispositivos, tales como: Pmods, micro-USB, HDMI, VGA, entre otros muchos [6].

Figura 3.5. Tarjeta Zedboard[10].

El dispositivo Zynq-7000 es un SoC que incorpora dos procesadores multinúcleo ARM Cortex-A9 junto con una FPGA de la serie 7 de 28 nm. De este modo, el sistema se divide en dos partes claramente diferenciadas, la parte del sistema de procesamiento, donde se ejecutará el software, y la parte de lógica programable, donde se generará el hardware modelado, quedando el sistema como se observa en la figura 3.6.

[10] Imagen extraída de: http://zedboard.org/product/zedboard

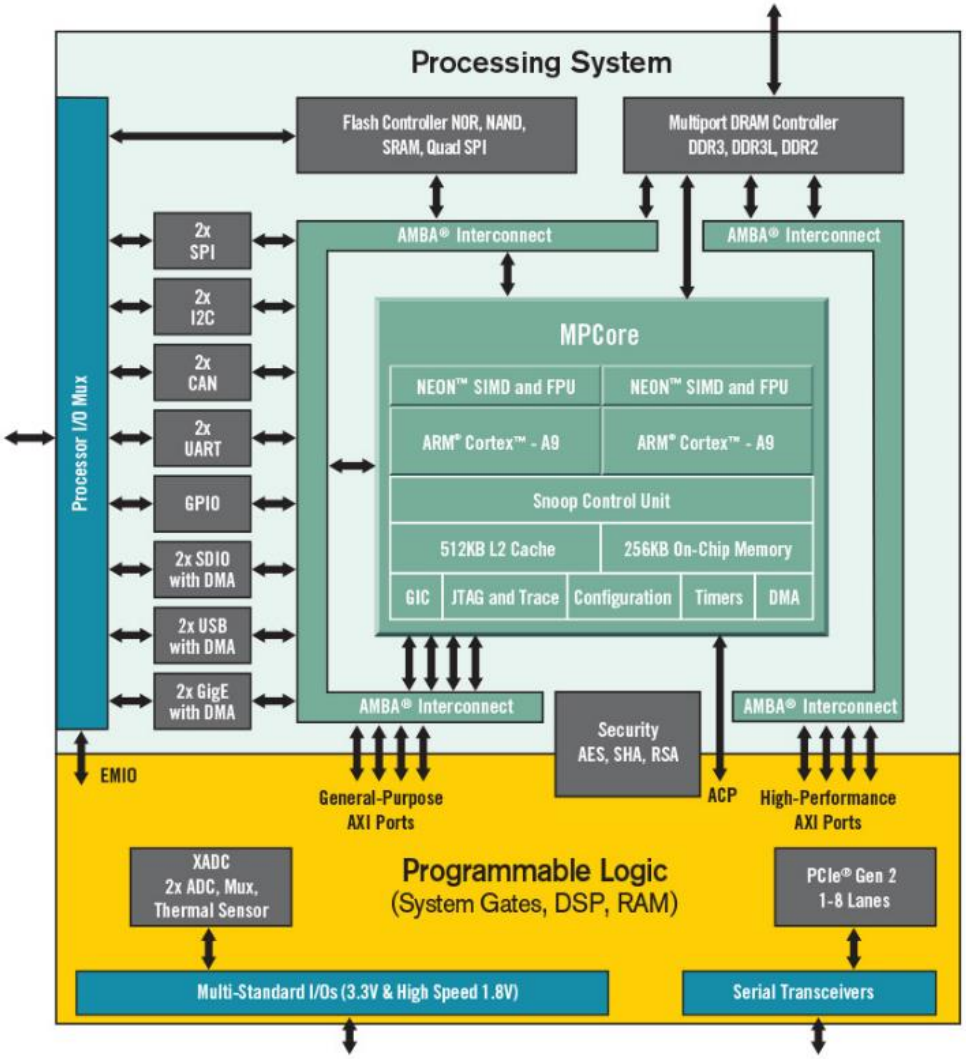


Figura 3.6. Arquitectura Zynq-7000 SoC[11].

3.4.1. Sistema de Procesamiento

El sistema de procesamiento o PS (*Processing System*) contiene en su interior un conjunto de bloques que forman una arquitectura similar a la de un microcontrolador. En este caso, está compuesto principalmente por la APU, interfaces de memoria, buses AMBA (*Advanced Microcontroller Bus Architecture*), diferentes periféricos y múltiples interfaces de entrada y salida para interactuar con el exterior.

El bloque central es el APU (*Application Processor Unit*), el cual está formado por dos procesadores ARM Cortex-A9, una memoria caché de 512 KB, una memoria ROM de 256KB, un controlador DMA (*Direct Memory Access*), el

[11] https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png

controlador de interrupciones y un conjunto de temporizadores, entre otras cosas, necesarias para el correcto funcionamiento del dispositivo.

Los buses AMBA son los encargados de comunicar los diferentes elementos que conforman el sistema de procesamiento, además de la comunicación de éste con la parte de lógica programable. Los diferentes protocolos que componen las especificaciones AMBA son los siguientes, los cuales dependen para que cometido se utilicen: AXI, AHB, ASB, APB.

Por último, el PS contiene los periféricos de entrada y salida que permiten comunicarse con elementos externos. La serie de periféricos que se encuentran en el dispositivo son los siguientes: USB, SDIO, GigE (*Gigabyte Ethernet*), SD, GPIO, UART, CAN, I2C y SPI. Estos periféricos no están conectados directamente a la interfaz externa, denominada MIO (*Multiplexed I/O*), ya que, debido a su reducido número de pines, es posible conectar los periféricos a través de la interfaz EMIO (*External Multiplexed I/O*), perteneciente a la parte PL. De este modo, es posible conectar periféricos del sistema de procesamiento con la lógica programable.

3.4.2. Lógica Programable

El bloque de lógica programable o PL (*Programable Logic*) puede diferir en distintos dispositivos de la familia Zynq-7000. La tarjeta Zedboard incluye el dispositivo Z-7020, compuesto por una FPGA Artix-7 formada por 8500 celdas lógicas, 106400 biestables, 53200 LUTs, 220 DSPs Slice y 4,9 Mb de RAM. En esta parte también se encuentra localizado el módulo XADC con dos ADCs de 12 bits con amplificadores, un multiplexor analógico de 17 canales, un sensor integrado de temperatura y de voltaje.

3.5. Interacción con el Sistema ULPS

Para realizar la conexión de la tarjeta Zedboard con las balizas que componen el sistema ULPS, son necesarios cincos conversores digitales-analógicos (DACs) [17], que transformen los códigos modulados guardados en el dispositivo digital en señales analógicas que se transmitan al medio en forma de señales ultrasónicas. Para ello, se han utilizado debido a su conexión a través de los puertos PMOD, los módulos PmodDA2, observados en la figura 3.7, desarrollados por Digilent.

Figura 3.7. PmodDA2 de Digilent[12].

Cada módulo contiene dos DACs DAC121S101 de Texas Instruments con una resolución de 12 bits cada uno, por lo que serán necesarios tres módulos para abordar los 5 DACs necesarios. Cada DAC121S101 funciona siguiendo el protocolo SPI.

3.6. Transductor Ultrasónico

La transmisión de las señales ultrasónicas del sistema ULPS se ha llevado a cabo mediante el transductor 328ST160 mostrado en la figura 3.8. Este dispositivo transmite en forma de ondas ultrasónicas las señales eléctricas analógicas que se observan a la salida de los DACs comentados anteriormente. Cuenta con un ancho de banda de 2,5 KHz, una frecuencia central de 25 KHz y una presión de sonido mínima de 115 dB.

Figura 3.8. Transductor ultrasónico 328ST160[13].

[12] Imagen extraída de: https://store.digilentinc.com/pmod-da2-two-12-bit-d-a-outputs/
[13] Imagen extraída de: https://es.farnell.com/prowave/328st160/transmisor-ultras-nicos-32-8khz/dp/1007330

Es necesario recalcar, que esta arquitectura propuesta proviene de una similar en la que, en lugar del módulo WiFi, existía un receptor WiFi que se conectaba con la Zedboard a través de una conexión ethernet. Por ello en el siguiente capítulo se explicará el diseño del sistema ULPS previo en el que se basa el presente trabajo, haciendo alguna alusión al protocolo ethernet. Aun así, no existen más diferencias en la arquitectura entre el sistema ultrasónico antiguo y el nuevo, ni existen cambios en la forma en la que las señales ultrasónicas son emitidas por parte del SoC.

Capítulo 4

Sistema Ultrasónico Previo

En este capítulo se explicará a nivel de funcionamiento software y hardware el sistema ULPS, en el cual se basa este proyecto. El sistema ULPS previo poseía una arquitectura similar a la de este proyecto con la excepción del módulo WiFi, que era sustituido por un router WiFi que iba conectado a la tarjeta Zedboard mediante una conexión ethernet y a través de la cual se realizaba la conexión con el PC externo. A partir de esto, el sistema será modificado para conseguir la comunicación WiFi mediante el módulo MRF24WG0MA y el ordenador con conexión WiFi y de esta manera reunir la arquitectura completa propuesta en el capítulo anterior. Aun así, el funcionamiento completo del sistema no se ve alterado por esta modificación de la arquitectura, por lo que el sistema que se explicará en este capítulo no se verá afectado para nada, al añadir el módulo WiFi. Para explicar todo este capítulo se ha hecho especial uso de un documento en el que se explica el sistema previo en el que se basa el presente proyecto [1].

4.1. Sistema Ultrasónico LPS

Partiendo de la explicación sobre sistemas ultrasónicos LPS realizado en el punto 2.1.1.4 del documento actual, se va a realizar una breve explicación del sistema ULPS en la que se basa este proyecto, al cual se le ha mejorado su arquitectura con la incorporación de un módulo WiFi que se explicará en el siguiente capítulo a éste.

Este sistema ULPS, teniendo en cuenta la arquitectura presentada en el capítulo anterior, presenta cinco balizas que periódicamente transmiten una señal modulada al medio. Por otro lado, un receptor que se mueva por el área de alcance del sistema puede detectar las cinco transmisiones ultrasónicas, determinando el TDOA (*Time Difference Of Arrival*), que junto a un algoritmo de posicionamiento sea capaz de estimar su propia posición dentro del espacio interior. Este sistema formado por las cinco balizas se puede replicar por un

amplio espacio interior, para que la suma de todos los sistemas haga que se tenga cobertura por todo el espacio.

Cada transductor ultrasónico, explicado en el punto 3.6, posee el amplificador, explicado en el punto 3.5, los cuales amplifican el voltaje proveniente del DAC, normalmente de 0 a 3.3 V, a un voltaje de salida que varía entre 0 y ± 12 V aplicado sobre los transductores.

El sistema ULPS completo, a cargo del sistema digital, cumple las siguientes características mostradas a continuación:

- El sistema digital ofrece una gran variedad para configurar diferentes señales trasmitidas con, a su vez, distintas longitudes. Esto permite que el sistema funcione en un mayor número de ambientes en los que se pueden dar distintos factores de complejidad, como niveles de ruidos e interferencias.
- Las balizas en las que se basa el proyecto pueden ser modificadas, lo cual implica un mayor número de posibles anchos de banda y frecuencia requeridos. Por ello, el sistema es capaz de adaptarse a distintos tipos de modulaciones con distintas frecuencias portadoras. Debido a esto, el sistema es capaz, no solo de generar modulaciones típicas basadas en fase y amplitud como BPSK (*Binary Phase-Shift Keing*), QPSK (*Quadrature Phase-Shift Keying*), ASK (*Amplitude Shift Keying*) o QAM (*Quadrature Amplitude Modulation*), sino también generar modulaciones multiportadoras más complejas, como FBMC (*Filter Bank Multiple Carrier*) o modulaciones como CDMA o TDMA para evitar las interferencias del tipo crosstalk. La frecuencia de muestreo de las señales tendrá un rango que varíe desde los 50 a los 500 kHz, que es lo máximo que permiten los DACs, haciendo que el diseño sea capaz de transmitir por cada baliza las señales a una frecuencia que permita que el sistema opere perfectamente en tiempo real.
- Como puede existir un gran número de unidades formadas por balizas en el ambiente, existe la opción de que se pueda configurar una unidad como maestra, para que ésta determine el comienzo de emisión de las siguientes y estar así todas sincronizadas.
- Por último, el sistema ofrece la posibilidad de cambiar la configuración de cada unidad en tiempo real a través de un enlace inalámbrico entre el sistema y un PC externo, permitiendo cambiar rápidamente la configuración de las balizas a favor de elegir la modulación que mejor se adapte a un determinado ambiente. Esto aporta una gran flexibilidad al sistema, pudiéndose modificar de una forma rápida cualquier configuración del sistema.

Todos estos puntos que cumple el sistema fueron desarrollados en el SoC Zynq, en el que a partir de una implementación hardware y software, que será descrita en el siguiente punto, se consiguió la correcta operación del sistema ULPS.

4.2. Diseño de la Arquitectura del SoC

En este punto se explicará el desarrollo hardware y software que se llevaron a cabo en el SoC para conseguir emitir las señales ultrasónicas de una forma correcta. Este sistema que se explicará a continuación es la base de este proyecto y se desarrollará para tener un conocimiento amplio del sistema ultrasónico completo que se verá modificado más tarde con la incorporación del módulo WiFi que sustituirá la conexión ethernet, sin contemplar esto un cambio significativo en el funcionamiento completo del sistema descrito en este capítulo.

Como cualquier aplicación realizada en un SoC, el desarrollo del sistema se separa en dos grandes fases de diseño. La primera de ellas consta del diseño hardware realizado sobre la lógica programable o FPGA del SoC, en el que se incorporan todos los elementos lógicos del sistema diseñados, como el periférico diseñado para manejar las cinco emisiones ultrasónicas. Y, por último, el desarrollo software que irá implementado en el sistema de procesamiento, gracias al ARM que posee, en el que se realizarán las tareas de más alto nivel, como la comunicación inalámbrica o las configuraciones del sistema.

Es importante señalar que todas las tareas llevadas a cabo en el SoC hubiesen sido posibles de realizar con una arquitectura basada únicamente en un procesador ARM. Aun así, la lógica programable proporciona flexibilidad y paralelismo al controlar las cinco balizas al mismo tiempo. Además, permite escalabilidad, ya que es posible extender el número de balizas del sistema de una forma rápida y sencilla sin necesidad de realizar modificaciones significativas al sistema. Existen trabajos previos a este proyecto que usaban un µcontrolador para realizar esto, viéndose enormemente limitado, debido a la gran cantidad de instrucciones que debían realizarse de forma secuencial para cumplir las funciones necesarias [8], realizándose únicamente las modulaciones más simples, y, por lo tanto, no llevarse a cabo técnicas de acceso al medio como CDMA o TDMA.

4.2.1. Diseño Hardware

El diseño hardware del sistema se centra en la elaboración de un periférico que permita controlar los DACs del sistema a fin de emitir los códigos modulados correctamente. La figura 4.1 representa el diagrama de bloques que se propuso para el periférico. En primer lugar, ya que los cinco DACs, explicados en el punto 3.5, presentan una interfaz SPI (*Serial Peripheral Interface*), el periférico contiene cinco controladores SPI maestros, uno por cada DAC, que han sido diseñados para trabajar a una frecuencia de 500 kHz, los cuales proporcionan los datos

siguiendo el protocolo SPI a los DACs. Dichos datos, a su vez, son entregados por cinco memorias dual-ports con un tamaño de 16kx16 bits, que están divididas cada una por dos bloques de 8kx16.

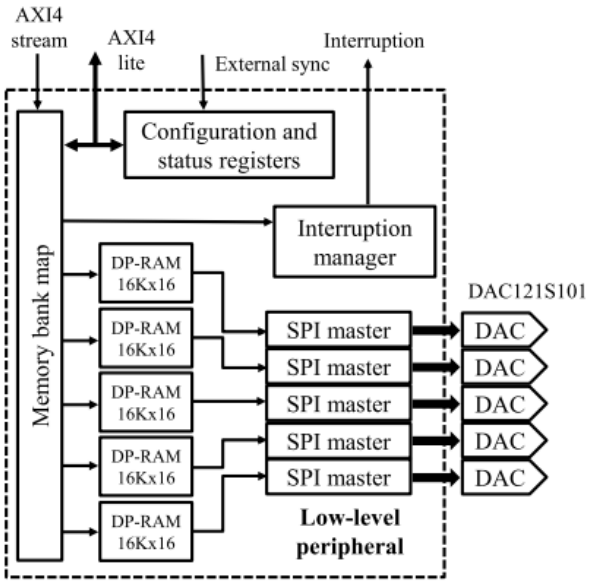


Figura 4.1. Diagrama de bloques propuesto para el periférico de control de las emisiones ultrasónicas[14].

El procesador ARM es responsable de guardar las señales a ser transmitidas por las balizas en las memorias dual-ports. Para ello, el procesador usa el puerto A de las memorias para escribir, empezando desde la primera posición de memoria, datos con un máximo de longitud de 16384 (16K) muestras de 16 bits cada una. El procesador accede al periférico a través de un bus AXI4-lite, en la que a través del decodificador de memoria mostrado en la figura 4.1 (Memory bank map), transfiere las señales desde el bus hasta las memorias de forma correcta. Por otro lado, el puerto B es leído por parte de los controladores SPI para obtener las muestras de las señales una vez que éstas son correctamente guardadas en las memorias dual-ports. El uso de estas memorias está justificado para que una vez que se configuren unas señales a emitir por parte del procesador ARM, este no tenga que volver a cargar estos datos en cada periodo de emisión, si no que tan solo sea necesario volver a leer las memorias desde la primera posición.

Para señales de una longitud mayor a la que es posible almacenar en las memorias dual-port, 16384 muestras, es necesario utilizar un banco de memoria DDR3 externo a la FPGA, ya que no es suficiente la memoria interna que posee. Este banco DDR3 se encuentra integrado en la tarjeta Zedboard, pudiendo acceder a él a través del procesador ARM del Zynq mediante otro bus AXI4-lite

[14] Imagen extraída de: Hernández A, Garcia E, Gualda D, Villadangos J.M., Nombela F, Ureñá J, "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*", IEEE transactions on instrumentation and measurement, vol. 66, no. 8, August 2017, pp. 1954-1964.

y poder así, guardar los datos que contienen en las memorias dual-port. Para llevar a cabo la transferencia en tiempo real de datos de la memoria DDR3 a las memorias dual-port es necesario hacer uso de una interrupción generada por parte del periférico para que el procesador ARM entre en una rutina de atención a dicha interrupción. Mientras las señales están transmitiéndose por las balizas, siempre que se hagan lecturas de bloques de 8kx16, que es el tamaño en el que se divide cada memoria dual-port, la interrupción es generada, permitiendo al ARM guardar las siguientes muestras de la señal encontradas en la memoria DDR3 y a su vez, a través del puerto B, que las muestras disponibles en el siguiente bloque de 8kx16 se han leídas por los controladores SPI. Con todo esto y teniendo en cuenta que el procesador ARM trabaja a una frecuencia de 650 MHz y el periférico a 100 MHz, es posible cumplir la condición de que los DACs trabajen a una frecuencia de muestreo de 500 kHz.

Para liberar al procesador ARM de carga de trabajo al querer emitir señales de más de 16384 muestras, se hizo uso de un módulo de acceso directo de memoria (DMA), para así llevar a cabo las trasferencias de memoria de la memoria DDR3 externa a las memorias dual-port integradas. Para ello, se usa un bus AXI4-stream, disponible también en el periférico, el cual es decodificado en el Memory Bank Map para transferir los datos a las memorias dual-port. En la figura 4.2 se muestra el diagrama global de la arquitectura del SoC en el que se encuentran las memorias DDR3 y el DMA.

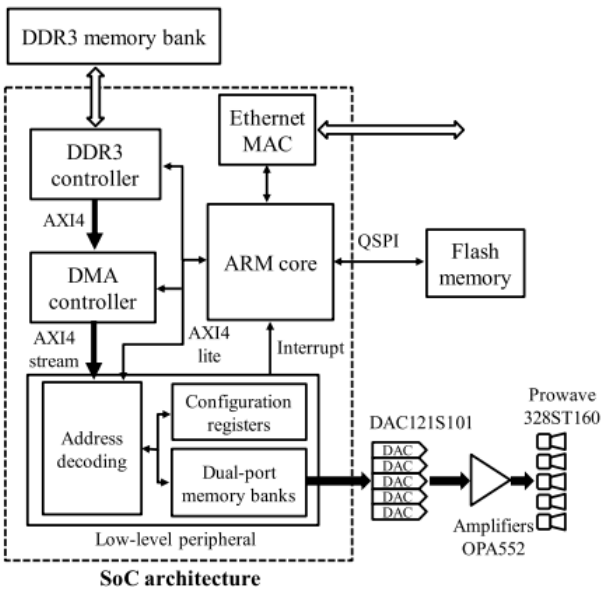


Figura 4.2. Diagrama de bloques de la arquitectura del SoC propuesto[15].

[15] Imagen extraída de: Hernández A, Garcia E, Gualda D, Villadangos J.M., Nombela F, Ureña J, "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*", IEEE transactions on instrumentation and measurement, vol. 66, no. 8, August 2017, pp. 1954-1964.

De cara a la configuración que determine la longitud de las señales y el periodo de repetición de éstas, existen un bloque de registros mapeados en el bus AXI4-lite, para que el ARM pueda acceder a ello para preparar la operación global del sistema. Además, existe una memoria flash externa, para que siempre que arranque el sistema se carguen tanto los parámetros como las propias señales a emitir. También hay un registro de estado para el periférico y su interrupción, usada para saber que bloque de 8k está disponible para direccionar las siguientes muestras de la señal.

Todo este desarrollo se llevó a cabo con Vivado, realizando el diseño del periférico en VHDL y añadiéndolo al diagrama de bloques generado, mostrado al completo en la figura 4.3. Se observa, además del módulo desarrollado, el sistema de procesamiento que contiene el bus de acceso a las memorias DDR, el DMA, el módulo de interconexión de los buses AXI, el módulo de reset del procesador y un módulo para concatenar las interrupciones del periférico diseñado y el DMA.

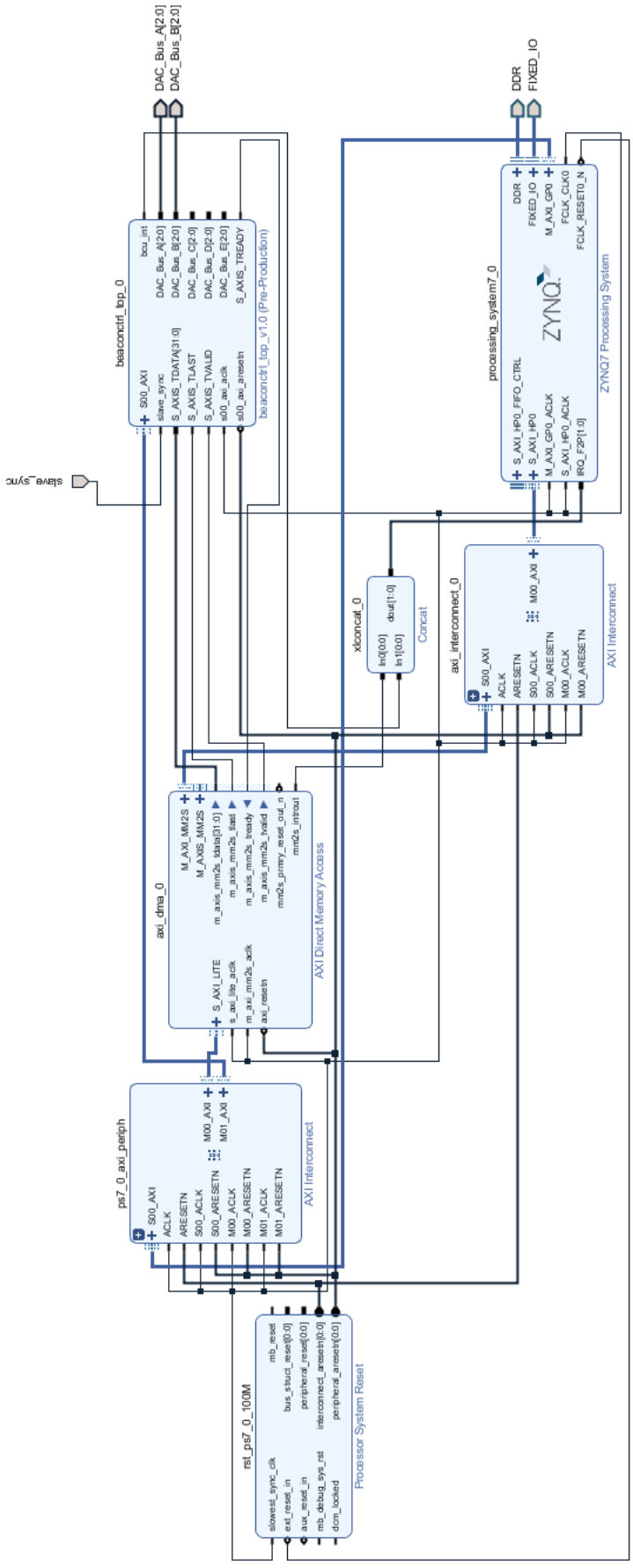


Figura 4.3. Diagrama de bloques de la arquitectura previa del sistema ULPS generada en Vivado.

4.2.2. Diseño Software

El procesador ARM es el encargado de inicializar todos los módulos una vez que arranque el sistema. Era responsable de configurar la interfaz ethernet, que permitía, a través de un receptor WiFi, tener conexión inalámbrica con un PC a través de un router WiFi. Esta conexión es utilizada para modificar las señales transmitidas, así como el periodo de repetición de las señales, a través de la aplicación que se mostraba en el punto 3.1 del presente documento. Si las señales que han de guardarse para la transmisión tienen una longitud mayor de 16384 muestras, el procesador ARM guarda éstas en el banco de memorias DDR3. Después, siempre que se produzca una interrupción desde el periférico, el procesador configura una trasferencia del DMA para cargar en las memorias dual-port el siguiente bloque de 8k datos. Aun con este método, la longitud de las señales está limitada a 128k muestras, aunque puede ser aumentado fácilmente con un bloque de memorias DDR3 de más tamaño.

Esta gran ventaja, dada por el DMA y el bloque de memorias DDR3, proporciona la oportunidad de que puedan trasmitirse señales con un gran número de muestras que permiten una gran variedad de señales moduladas, pudiendo utilizar la que mejor funcione en cada medio donde el sistema ULPS esté instalado.

Además, gracias a la memoria flash externa al SoC Quad SPI, es posible guardar las señales y las configuraciones, para que, al arrancar el sistema, éste no necesite la conexión inalámbrica con el PC externo para funcionar. De esta manera es posible recuperar la última configuración del sistema y que éste empiece a trabajar de la misma forma que lo hizo la última vez configurado.

Teniendo todo lo descrito en cuenta, en la figura 4.4 se muestra un flujograma con la secuenciación del código que describe la rutina principal del procesador. Después de una fase en la que se inicializan la plataforma del Zynq, el timer, la interrupción, el link de ethernet y el módulo DMA, el procesador accede a la memoria flash para leer los parámetros configurados por última vez y configurar el sistema con estos datos. A partir de este momento, el ARM lleva a cabo un proceso que controla la llegada de un nuevo mensaje ethernet de configuración, que se explicará en el punto 5.2.5.3 del siguiente capítulo. Y si es así, guardar la información tanto en la memoria flash como en la memoria DDR3, si la longitud de las señales excede las 16384 muestras.

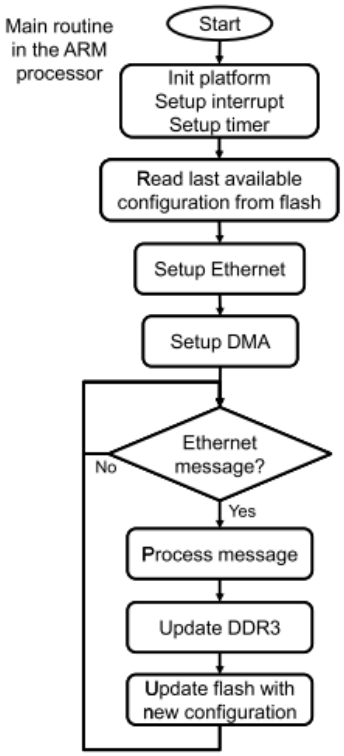


Figura 4.4. Flujograma de la rutina principal que sigue el microprocesador ARM[16].

Por otro lado, en la figura 4.5, se muestra el servicio de rutina de interrupción generado por el periférico al haberse leído 8k muestras por parte del controlador SPI en señales de más de 16384 muestras. Esta rutina, en primer lugar, verifica el estado del periférico para confirmar la fuente de la interrupción. Después determina el actual bloque de 8kx16 datos para recibir las siguientes muestras, así como el área donde las muestras de las señales están guardadas en la memoria DDR3. A partir de esto, el procesador configura el DMA para la correcta transferencia de datos, para cargar las siguientes muestras de las señales en las memorias dual-port del periférico.

[16] Imagen extraída de: Hernández A, Garcia E, Gualda D, Villadangos J.M., Nombela F, Ureñá J, "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*", IEEE transactions on instrumentation and measurement, vol. 66, no. 8, August 2017, pp. 1954-1964.

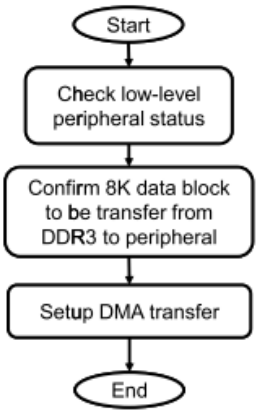


Figura 4.5. Flujograma del servicio de atención a la interrupción generada por el periférico[17].

Todo el código explicado en esta sección está incorporado en el Anexo del presente documento, en el que la rutina principal corresponde con la función *main()* y la rutina de atención a la interrupción a la función *BCUISR()*, viéndose la primera alterada mínimamente al haber eliminado el código correspondiente a la comunicación ethernet, por la función *StartWIFI_Aplication(),* que es una máquina de estados que se explicará en el punto 5.2.5.2 utilizada para la conexión cliente-servidor del módulo WiFi con el PC a través del router WiFi, y la función de inicialización de ethernet por la función, *PmodWIFI_Initialize()*, encargada de inicializar las direcciones de memoria necesarios para controlar el módulo WiFi. También se ha visto modificado al eliminar un timer que se llevaba a cabo en el fichero platform.c, necesario para la pila proporcionada por Xilinx para realizar la conexión ethernet correctamente.

Una vez explicado el sistema previo del que parte el proyecto, se explicará, en el siguiente capítulo, la mejora propuesta del sistema ULPS, que como se ha comentado anteriormente, consiste en la incorporación del módulo WiFi explicado en el punto 3.2, para que éste cumpla el mismo propósito que el receptor WiFi basado en ethernet previo, que consistía en la configuración, a través de un enlace inalámbrico, del sistema ULPS. También se completará toda la información correspondiente a la configuración inalámbrica del sistema que no ha llegado a abarcarse en este capítulo por completo.

[17] Imagen extraída de: Hernández A, Garcia E, Gualda D, Villadangos J.M., Nombela F, Ureñá J, "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*", IEEE transactions on instrumentation and measurement, vol. 66, no. 8, August 2017, pp. 1954-1964.

Capítulo 5

Mejora del Sistema Ultrasónico

El sistema previo, tal y como se ha comentado en el capítulo anterior, realizaba la comunicación con un PC externo a través de una conexión ethernet conectada a un router WiFi. Con la llegada de módulos WiFi de bajo coste es posible establecer conexión, sin necesidad de conexiones físicas que requieren de una gran estructura, permitiendo así, una conexión directa, que puede llevarse a cabo dentro de una subred WiFi, en la que el punto de acceso o router esté situado a varios metros de distancia, tal y como puede estar configurada una subred inalámbrica en los hogares.

Para poder incorporar el módulo WiFi basado en el chip MRF24WG0MA a la arquitectura del sistema, se ha hecho uso de un periférico diseñado por Digilent, que se ha incorporado a la lógica programable del SoC. Aunque el chip está preparado para funcionar directamente con un microcontrolador, como sería en este caso, el ARM que incorpora el Zynq es de gran utilidad hacer uso del hardware configurable que aporta el SoC para hacer de manera eficiente, flexible y paralela muchas de las tareas necesarias para generar la comunicación del módulo WiFi con la plataforma Zedboard.

Por ello, la incorporación del módulo WiFi necesita una implementación hardware, sobre la lógica programable en la que se incorporará un periférico, y un desarrollo software, implementado en el sistema de procesamiento, sobre él que se incorporará la pila de protocolos TCP/IP y se realizarán tareas de más alto nivel, como la configuración de la conexión WiFi o la configuración del sistema, entre otras cosas que se explicarán a continuación. Para ambas partes del desarrollo, ha sido de gran utilidad el código proporcionado por Digilent, tanto en la incorporación el periférico como en la pila de protocolos TCP/IP [16]

5.1. Implementación Hardware

Como se ha descrito anteriormente, el módulo WiFi no va conectado directamente al microcontrolador que posee el SoC, sino que se ha hecho uso

de un diseño hardware propuesto por Digilent Inc., que irá incorporado en la lógica programable. De esta manera, no será necesario utilizar los elementos que incorpora la unidad de procesamiento para la comunicación con el módulo WiFi.

5.1.1. Funcionamiento del Circuito MRF24WG0MA

Aunque el diseño hardware del periférico ofrecido por Digilent Inc. no ha sido necesario desarrollarlo, se explicará su funcionamiento a continuación, ya que el software de bajo nivel utilizado para la comunicación con el dispositivo está basado en la existencia de este periférico incorporado en el diseño. Para ello es necesario conocer todo lo relativo al chip MRF24WG0MA para que pueda ser utilizado correctamente.

5.1.1.1. Conexionado con la Tarjeta de Desarrollo.

El chip MRF24WG0MA debe estar conectado con la Zedboard de la manera presentada en la figura 5.1, en la que se observan las señales que han de generarse para llevar a cabo la comunicación entre los dos dispositivos. Las señales de más importancia en la comunicación vienen dadas por el protocolo SPI, que necesita para su funcionamiento las señales CS, SDI, SDO y SCK. Además, es necesario una señal de interrupción activada por el chip MRF24WG0MA y las señales de propósito general para activar o desactivar las entradas de HIBERNATE, RESET o WP del chip.

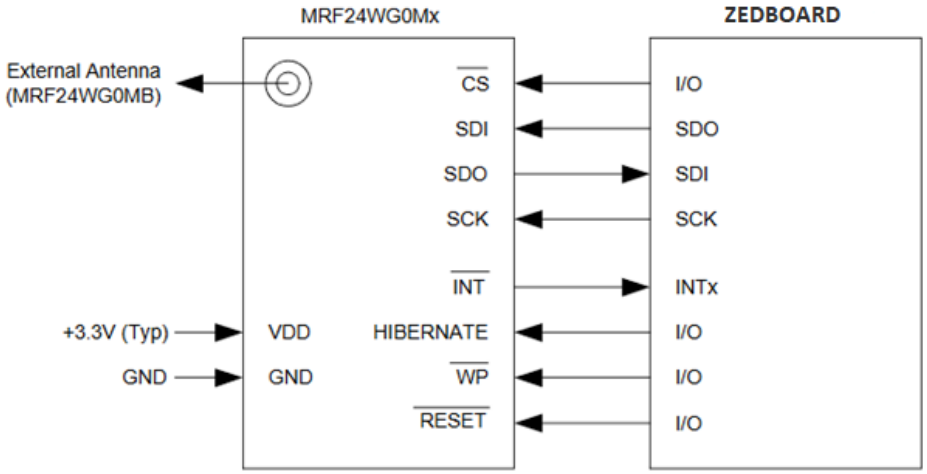


Figura 5.1. MRF24WG0MA conectado a Zedboard[18].

[18] Imagen extraída de: http://ww1.microchip.com/downloads/en/DeviceDoc/70686B.pdf

5.1.1.2. Interfaz SPI

La interfaz serie SPI es usada para transmitir los datos entre el Zynq-7000 y el módulo MRF24WG0MA, teniendo en cuenta que el módulo WiFi es el esclavo y el SoC el maestro. Este periférico trabaja junto con la línea de interrupción (INT), ya que cuando hay datos disponibles para el maestro durante la operación, el módulo MRF24WG0MA cambia a nivel bajo la señal de interrupción y después de que se transfieran los datos, cambia el valor a nivel alto. Por otro lado, el pin CS debe alternarse con bloque de datos transferidos, no pudiendo mantenerse en nivel bajo permanentemente. El cambio a nivel bajo se usa para indicar el inicio de una transferencia, y el cambio a nivel alto para indicar una transferencia completa. Todo esto se puede observar en la figura 5.2, en la que se muestran los tiempos mínimos de operación, tanto en la transmisión como en la recepción de datos.

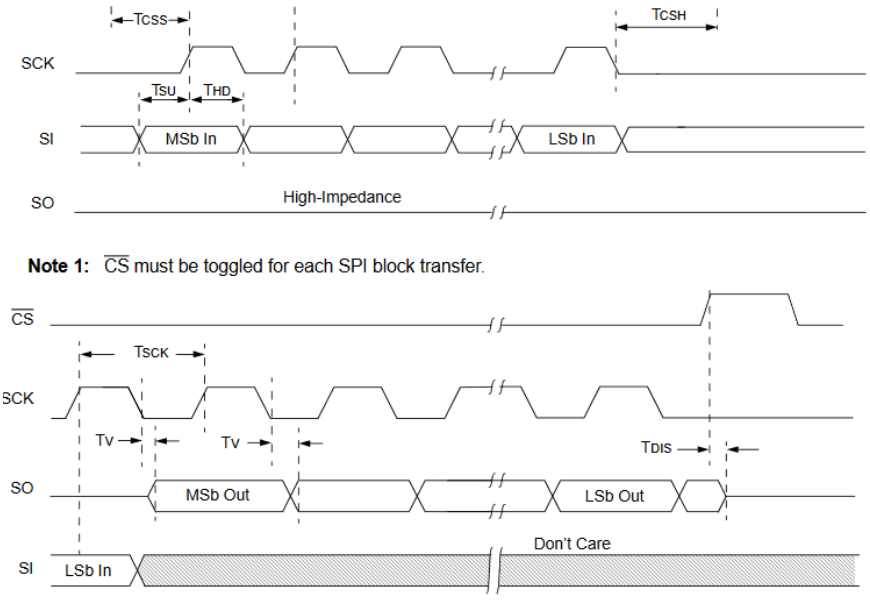


Figura 5.2. Señales SPI entre el chip MRF24WG0MA y el Zynq[19].

5.1.1.3. Estados de Energía

Es importante conocer, también, los estados de energía en los que se puede encontrar el módulo WiFi. El estado Hibernate es el estado más cercano al apagado que el dispositivo puede alcanzar; se controla a través del pin HIBERNATE, que, al ponerse en nivel alto, lleva al módulo a este estado

[19] Imagen extraída de: http://ww1.microchip.com/downloads/en/DeviceDoc/70686B.pdf

proporcionando así una posible mayor duración de la batería del dispositivo maestro, en este caso, la Zedboard. El estado Power Save (PS) es el que se alcanza automáticamente cuando no existe transferencia de datos; si algún tráfico aparece en la cola de datos del módulo, éste se despertará y obtendrá los datos del punto de acceso. De este modo se permite que el dispositivo esté sólo totalmente encendido cuando sea necesario, haciendo un uso más razonable de la energía. Por último, se encuentra el estado activo, en el que se encuentra el módulo totalmente encendido para la correcta transmisión y recepción de datos.

5.1.2. Periférico WiFi

En la figura 5.3 se muestra la interfaz gráfica o IP Core del periférico incorporado al diseño de bloques. Se pueden observar las señales de entrada principales, AXI_LITE_SPI, S_AXI_TIMER, AXI_LITE_WFCS y AXI_LITE_WFGPIO, que recogen su nombre de los distintos cuatro módulos IP que forman a su vez el periférico mostrado. También posee la señal de entrada de reloj, s_axi_aclk, y la señal de reset activa a nivel bajo, s_axi_arestn, ambas comunes a todos los bloques del sistema. Las salidas son Pmod_out, que es la salida conectada a la interfaz Pmod de la Zedboard, a la que va conectada el módulo WiFi con las conexiones ya descritas, y la interrupción WF_INTERRUPT conectada al sistema de procesamiento.

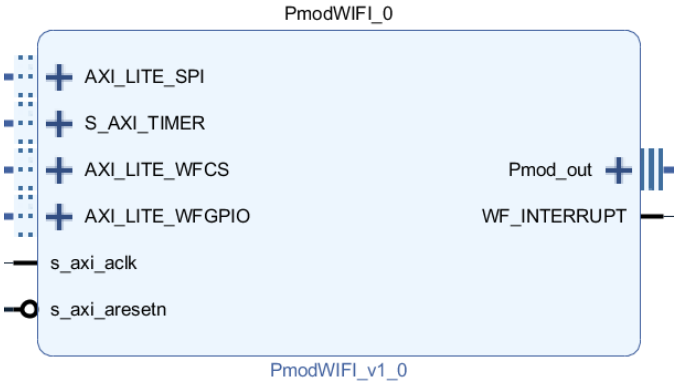


Figura 5.3. Periférico que implementa la comunicación con el módulo WiFi.

5.1.2.1. Módulos del Periférico

Como se ha comentado anteriormente, el periférico está formado por cuatro módulos IP, en los que se encuentra un AXI_SPI, AXI_TIMER y dos AXI GPIO, llamados WFCS y WFGPIO, además de un PMOD_BRIDGE. Éstos, a través de las señales AXI y el bloque de conexión AXI, AXI_Interconnect, que se encuentra fuera del módulo, consiguen implementar los siguientes periféricos como si cada uno fuera una dirección de memoria.

El periférico AXI SPI se utiliza para la generación de la señal SPI en la cual se basa la comunicación con el chip. El módulo AXI GPIO, llamado con el mismo nombre, se encarga de generar las señales de entrada reset e hibernate encargadas, entre otras cosas, de mediante una secuencia determinada, arrancar el circuito MRF24WG0MA. Este módulo también se encarga, mediante la habilitación de la interrupción que tiene asociada, de que cuando el circuito genere una interrupción en su señal de salida INT, el periférico a su vez genere mediante su otra señal WF_INTERRUPT, otra interrupción que llegue al procesador ARM y este haga saltar a una rutina de atención a la interrupción. La interrupción dada por el circuito MRF24WG0MA se produce para informar de que existen nuevos datos encolados para ser procesados por el SoC provenientes del punto de acceso. De esta manera la rutina de atención a la interrupción se encarga a nivel software, mediante la función *WF_EintHandler (),* encontrada en el fichero wf_eint.c, de leer los datos registrados en el circuito y transferirlos mediante el controlador SPI al SoC.

El otro módulo AXI GPIO, llamado CS debido a la función que realiza, es el encargado de activar o desactivar la señal Chip Select (CS) y de esta manera habilitar o no el circuito. Por último, el temporizador AXI TIMER, aunque no es utilizado para generar ninguna señal de conexión con el circuito, es utilizado para que el propio procesador ARM pueda disponer de un temporizador de 1 ms, sin necesidad de utilizar los temporizadores que contiene, destinado a realizar labores necesarias para manejar el circuito MRF24WG0MA, tales como manejar el tiempo necesario en el arranque completo del circuito o controlar el tiempo en el que un paquete de respuesta llega al circuito. El periférico incorpora también un Pmod Bridge usado para conectar los distintos controladores AXI que componen el periférico con la interfaz Pmod de salida que se conecta al Pmod WiFi de Digilent que contiene el circuito.

5.1.2.2. Conexionado con la Unidad de Procesamiento

Para acceder desde la unidad de procesamiento a cada uno de los periféricos que se encuentran incorporados en el módulo, basta con habilitar un puerto maestro AXI de propósito general, tal y como estaba realizado para acceder al periférico que controla el sistema ULPS y al DMA. De esta manera, a través de un AXI_INTERCONECT que permite cualquier combinación de dispositivos maestro o esclavos AXI con diferentes anchos de datos, reloj y sub-protocolos AXI, se redireccionan los datos AXI de forma apropiada desde el sistema de procesamiento hasta los periféricos que componen el sistema, en este caso, el DMA, el periférico ULPS y las entradas de SPI, de TIMER y de GPIO del periférico WiFi. Así, para acceder a dichos módulos, basta transmitir los datos desde el puerto maestro AXI del sistema de procesamiento, con la dirección de memoria del módulo al que se quiera acceder, lo cual se realiza a nivel software, en las capas que tienen interacción con el hardware, lo que se conoce como BSP (*Board Support Package*).

5.1.3. Incorporación del Periférico WiFi a la Arquitectura

En primer lugar, para poder reutilizar la arquitectura hardware del anterior sistema, únicamente se ha generado un nuevo proyecto en la plataforma Vivado en el que se han eliminado los recursos del sistema de procesamiento que ya no se utilizan, siendo en este caso únicamente el módulo ethernet, para llevar a cabo la comunicación por este medio físico.

Una vez realizado esto, basta añadir el nuevo periférico al diseño de bloques. Esto se puede realizar de distintas maneras, o bien seleccionando el botón en la ventana del diseño de bloques de añadir IP y añadir el módulo, tal y como se muestra en la figura 5.4, o añadiéndolo a partir de la pestaña boards, mostrado en la figura 5.5.

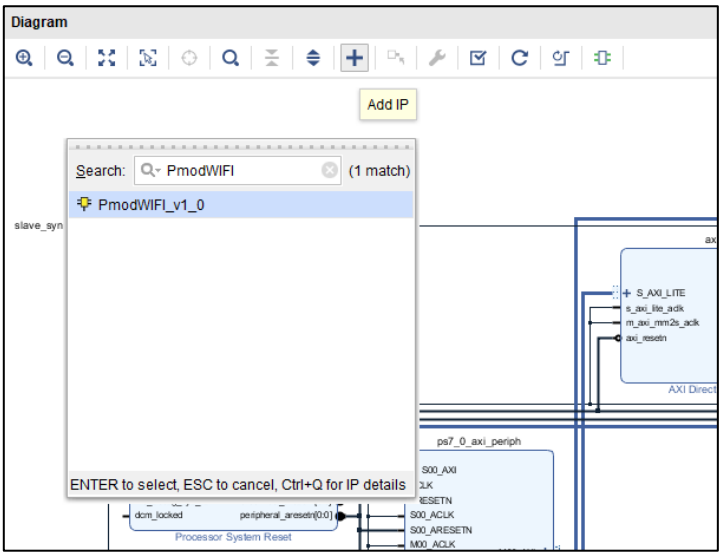


Figura 5.4. Añadir periférico con la función Add IP de Vivado.

En este caso, se ha utilizado la segunda opción, ya que de esta forma se permite elegir el puerto de la Zedboard a utilizar, realizando la asignación automática de los pines del Zynq-7000 con los puertos de la tarjeta, sin necesidad de especificarlos en el fichero system.xdc utilizado para realizar dicha asignación. Cabe destacar, que para realizar esto es necesario añadir en la creación del proyecto, un fichero de restricciones (Constraints) que contemple la asignación de los pines de la placa, en este caso, la Zedboard.

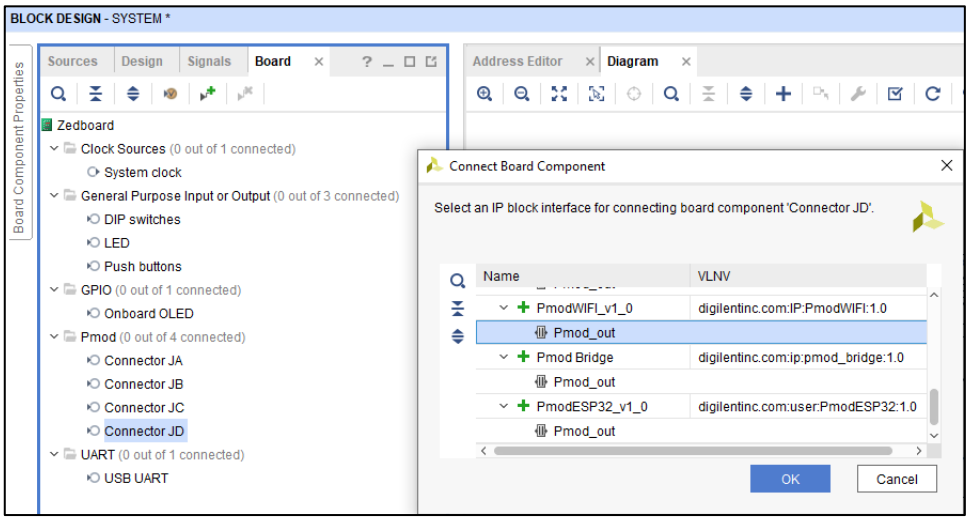


Figura 5.5. Añadir el periférico a través de la pestaña Boards.

Una vez añadido de esta manera, se escoge la opción de realizar el conexionado de las señales automáticamente, que permite conectar todas las señales del periférico y ampliar el AXI_INTERCONNECT con las señales de salida que se dirigen hacia él, quedando el sistema completo como se muestra en la figura 5.7, en el que al sistema ya realizado se le ha añadido el módulo WiFi, perfectamente conexionado para ser utilizado a nivel software a través de la unidad de procesamiento del SoC. Por último, antes de realizar la síntesis e implementación de la arquitectura, es necesario realizar un apropiado direccionamiento de todos los periféricos que componen el sistema, establecido el rango de memoria que cada uno debe poseer. El direccionamiento utilizado es el mostrado en la figura 5.6, en el que se observan también las direcciones de todos los periféricos, necesarios para acceder a cada uno de ellos a través del software.

Address Editor | Diagram | system.xdc

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
beaconctrl_top_0	S00_AXI	S00_AXI_reg	0x43C0_0000	512K	0x43C7_FFFF
PmodWIFI_0	AXI_LITE_SPI	Reg0	0x43C8_0000	64K	0x43C8_FFFF
PmodWIFI_0	S_AXI_TIMER	Reg0	0x43C9_0000	64K	0x43C9_FFFF
PmodWIFI_0	AXI_LITE_WFCS	Reg0	0x43CA_0000	64K	0x43CA_FFFF
PmodWIFI_0	AXI_LITE_WFG...	Reg0	0x43CB_0000	64K	0x43CB_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

Figura 5.6. Direccionamiento de memoria del SoC Zynq-7000.

Cabe destacar, por tanto, que el periférico desarrollado por Digilent se encarga de generar estas señales sin necesidad de utilizar los recursos del sistema de procesamiento que incorpora el Zynq-7000. Esto es así debido a la gran versatilidad que ofrece el poder usar este mismo diseño para distintos SoC o FPGAs en las que se utilice un microcontrolador, como podría ser un MicroBlaze comúnmente usado. De esta manera, también, se ha podido hacer uso del software de bajo nivel proporcionado por Digilent Inc., el cual desarrolla el acceso a dicho periférico para poder comunicarse con el módulo WiFi, de una manera transparente y sencilla para el desarrollador que quiera utilizar dicho producto, que, junto con la pila de protocolos TCP/IP proporcionada, no necesita realizar el desarrollo de drivers para utilizarlo, si no únicamente desarrollar la aplicación TCP/IP haciendo llamadas a funciones proporcionadas por dicha pila, abstrayéndose de su funcionamiento de más bajo nivel.

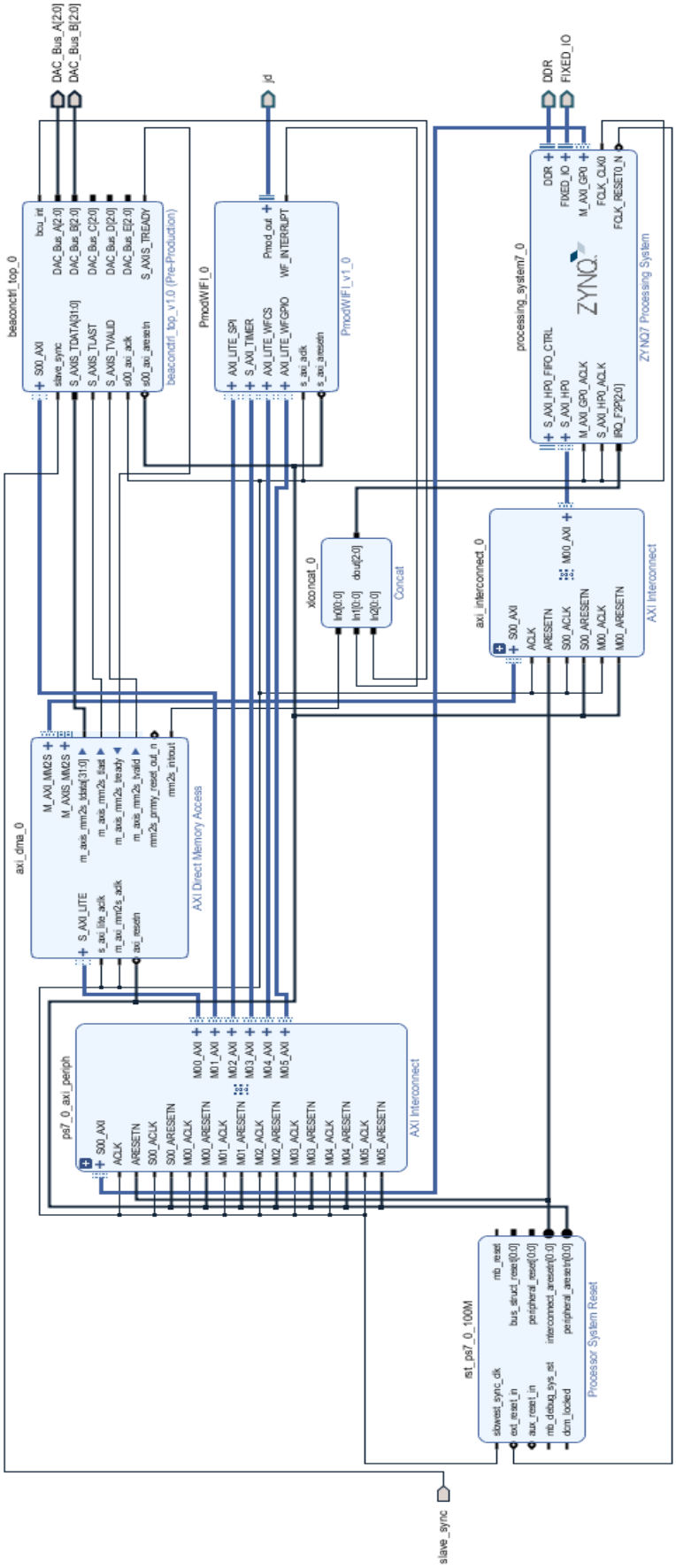


Figura 5.7. Modelo hardware completo del sistema.

5.2. Implementación Software

El software, además de realizar todas las funciones que se vieron en el apartado 4.2.2., va a llevar a cabo la comunicación a través del módulo WiFi. Por un lado, se van a realizar las tareas de más bajo nivel, que corresponden con la comunicación y el control del módulo WiFi con el Zynq para que funcione correctamente; y por otro, que es el que concierne al desarrollo software de este proyecto, la transmisión y recepción de datos entre el PC bajo el que corre el programa que configura el sistema ULPS y el módulo WiFi, todo ello con una conexión a través del punto de acceso.

5.2.1. Exportación del Modelo Hardware

Para llevar a cabo la implementación software, es necesario, en primer lugar, exportar el diseño hardware creado con la herramienta Vivado en la otra herramienta de Xilinx, SDK (*Xilinx Software Development Kit*), para de esta manera ser capaz de cargar un programa en el sistema de procesamiento y utilizar, de este modo, el diseño hardware realizado.

Para ello, basta con exportar el diseño hardware a través de la pestaña que se observa en la figura 5.8 y automáticamente se crea un modelo que puede ser usado por una plataforma software, en este caso SDK. Este modelo exportado contiene todo lo necesario para utilizar los periféricos añadidos a la lógica programable del SoC. El periférico WiFi, por ejemplo, contiene todos los drivers necesarios para su uso, así como la pila TCP/IP que utiliza.

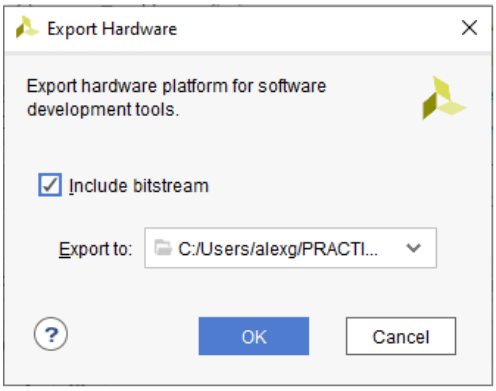


Figura 5.8. Pestaña para exportar el hardware modelado.

Una vez exportado, solo queda crear el proyecto software principal y añadir el BSP (*Board Support Package*), que se genera automáticamente, el cual incluye los drivers del diseño hardware, además de las librerías añadidas en el modelo hardware de los periféricos. A parte, para poder grabar el programa, ya sea en

memoria Flash o a través de una tarjeta SD introducida en la Zedboard, es necesario crear un proyecto Bootloader, encargado de arrancar el programa una vez que se haya grabado en una de las formas ofrecidas. De esta manera, la estructura del proyecto queda como se observa en la figura 5.9, en la que se observan el proyecto principal relacionado con la aplicación, BCU_top; el Bootloader, BCU_FSBL; y sus respectivos proyectos BSP, BCU_FSBL_bsp y BCU_top_bsp, todos ellos bajo el proyecto que contiene el hardware modelado, llamado SYSTEM_wrapper_hw_platform_0.

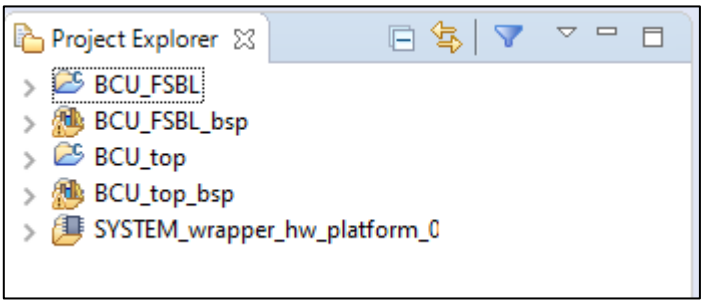


Figura 5.9. Distintos proyectos creados para generar el software.

5.2.2. Organización del Software

Las dos tareas diferenciadas que realiza el software, en este caso respecto al módulo WiFi, se organizan en diferentes secciones para identificar su labor. El software de más bajo nivel o drivers se encuentra en el BSP, llamado así porque es el que está en contacto directo con el hardware modelado y en el que se encuentra, además, la pila TCP/IP bajo el que se regirá la conexión entre los distintos puntos, aunque no esté realmente en contacto directo con el hardware. Por otro lado, se encuentra la aplicación propiamente dicha, que se encarga de llevar a cabo la transmisión y recepción de los datos. Esto se puede observar en la figura 5.10, en la que se distinguen las distintas capas del modelado software relativo al módulo WiFi, en el que el BSP abarca los drivers y la pila TCP/IP.

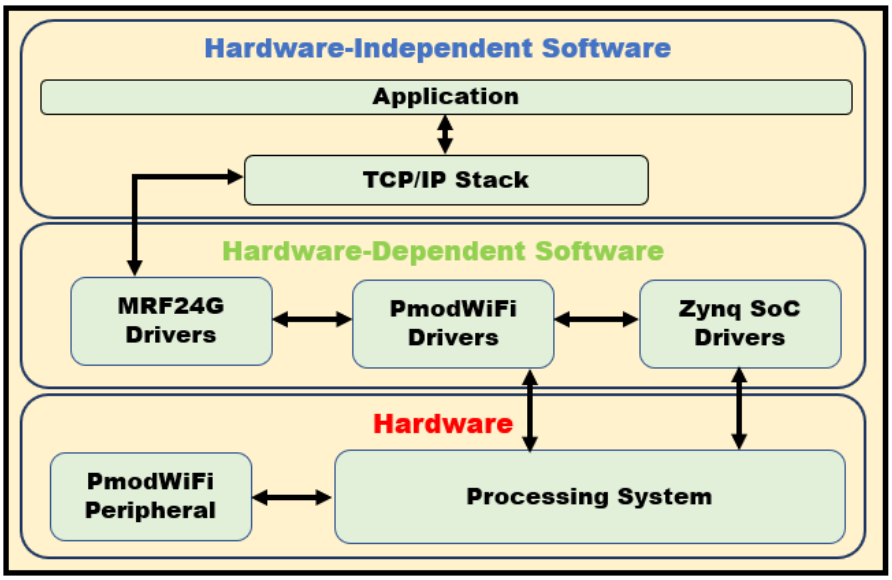


Figura 5.10. Capas software que contiene el proyecto.

5.2.3. Drivers del Módulo WiFi

El software de más bajo nivel del módulo WiFi está en las capas del BSP generado automáticamente al incluir en el proyecto de SDK, el hardware bajo el que va a correr dicho programa. Esto es debido a que el periférico que se utiliza para la comunicación con el módulo tiene incorporado dicho software en sus ficheros, lo que hace que automáticamente se añada al BSP, sin tener que realizar nada.

Este BSP relativo al módulo WiFi incluyen una pila TCP/IP que se utiliza para implementar la conexión en la red inalámbrica formada por el PC, el punto de acceso y el módulo. También se realiza el control del módulo, para que funcione correctamente a través del periférico que se utiliza, enviando los valores adecuados a través de las señales AXI, que comunican el procesador con el periférico, para que éste produzca las señales que se describieron en el apartado 5.1.2.2. Todo lo que incluye el BSP relativo al módulo WiFi se observa en la figura 5.11, en la que se observan todos los campos que cubren dichos drivers, de más bajo a más alto nivel respecto a su interacción con el hardware.

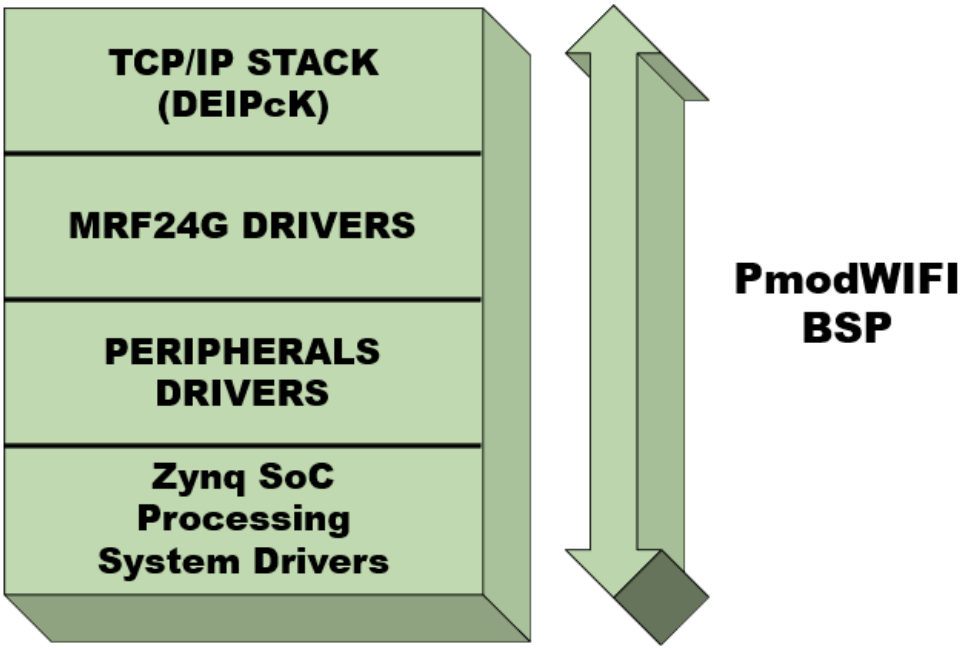


Figura 5.11. Contenido del BSP relativo al módulo WiFi.

5.2.3.1. Drivers del Zynq SoC

En primer lugar, se encuentran en el BSP los ficheros correspondientes a los drivers de inicialización del sistema de procesamiento del Zynq, que facilitan la habilitación necesaria de elementos del sistema de procesamiento como DDR, clocks, phase-locked loops (PLLs), MIOs y periféricos del sistema.

Por otro lado, se encuentran los ficheros que contienen las direcciones de memoria que se asignaron durante la implementación de la arquitectura hardware a los periféricos de la FPGA, a los cuales se accede a través del sistema de procesamiento mediante sus interfaces AXI. Conociendo dichas direcciones se puede acceder al periférico WiFi, únicamente escribiendo los datos necesarios para controlar dicho periférico en esa dirección del mapa de memoria. En el código presente en la figura 5.12 se puede observar las direcciones de memoria del periférico WiFi para poder controlarlo, generadas en el fichero xparameters.h.

```
/* Definitions for driver PMODWIFI */
#define XPAR_PMODWIFI_NUM_INSTANCES                 1

/* Definitions for peripheral PMODWIFI_0 */
#define XPAR_PMODWIFI_0_DEVICE_ID 0
#define XPAR_PMODWIFI_0_AXI_LITE_SPI_BASEADDR       0x43C80000
#define XPAR_PMODWIFI_0_AXI_LITE_SPI_HIGHADDR       0x43C8FFFF
#define XPAR_PMODWIFI_0_AXI_LITE_WFCS_BASEADDR      0x43CA0000
#define XPAR_PMODWIFI_0_AXI_LITE_WFCS_HIGHADDR      0x43CAFFFF
#define XPAR_PMODWIFI_0_AXI_LITE_WFGPIO_BASEADDR    0x43CB0000
#define XPAR_PMODWIFI_0_AXI_LITE_WFGPIO_HIGHADDR    0x43CBFFFF
#define XPAR_PMODWIFI_0_S_AXI_TIMER_BASEADDR        0x43C90000
#define XPAR_PMODWIFI_0_S_AXI_TIMER_HIGHADDR        0x43C9FFFF


/******************************************************************/
```

Figura 5.12. Definiciones de las direcciones de memoria del periférico WiFi.

5.2.3.2. Drivers del Periférico WiFi

Las direcciones de memoria comentadas anteriormente son utilizadas por los drivers del periférico WiFi para controlar los módulos que a su vez lo componen. En la figura 5.13 se puede observar el flujograma de la función *WF_SpiInit()*, mostrada también en la figura 5.14, encargada de inicializar el módulo SPI del periférico WiFi haciendo uso de dichas direcciones. Esta función, en primer lugar, utiliza el módulo de propósito general (GPIO) del periférico WiFi, llamado CS y encargado de manejar la entrada Chip Select (CS) del circuito MRF24WG0MA, para escribir un nivel alto en ésta y deshabilitar así el circuito. En segundo lugar, configura el módulo CS para que únicamente funcione como salida y no se puedan leer datos desde él. Estas dos instrucciones se realizan debido a que la entrada Chip Select (CS) del circuito se controla desde este módulo GPIO y no desde el controlador SPI. Por último, se configura y se habilita el controlador SPI que maneja los datos transferidos al circuito.

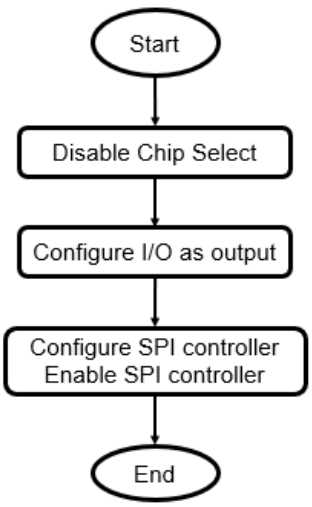


Figura 5.13. Flujograma de la función de inicialización del módulo SPI que contiene el periférico WiFi.

```
void WF_SpiInit(void)
{
    // The MRF24WG chip select line will be controlled using
    // an I/O pin, not by the SPI controller.

    // set the level high (chip select disabled)
    Xil_Out32(WF_CS_BASEADDRESS, 1);
    // Configure the I/O pin as an output and drive it
    Xil_Out32(WF_CS_BASEADDRESS+4, 0);

    // Configure SPI1:
    //   * Master Mode enabled (MSTEN = 1)
    //   * Clock Polarity is idle high, active low (CKP = 1)
    //   * Serial output data changes on transition from idle
    //     to active, or high to low transition (CKE = 0)
    //   * Data sampled at end of output time (SMP = 1), on rising edge

    // enable SPI controller
    WFSPIConfig.BaseAddress = WF_SPI_BASEADDRESS;
    XSpi_CfgInitialize(&WF_Spi, &WFSPIConfig, WF_SPI_BASEADDRESS);
    XSpi_SetOptions(&WF_Spi, XSP_MASTER_OPTION  |
                    XSP_MANUAL_SSELECT_OPTION   |
                    XSP_CLK_ACTIVE_LOW_OPTION   |
                    XSP_CLK_PHASE_1_OPTION);
    XSpi_SetSlaveSelect(&WF_Spi, 1);
    XSpi_Start(&WF_Spi);
    XSpi_IntrGlobalDisable(&WF_Spi);
}
```

Figura 5.14. Función de inicialización del módulo SPI que contiene el periférico WiFi.

Está función se encuentra en el fichero wf_spi_stub.c y se puede observar que las direcciones de memoria de los módulos SPI y CS del periférico, no se llaman exactamente igual. Esto es debido a que se han copiado los valores de las direcciones en otras variables al inicializar el software del módulo WiFi como se muestra en la figura 5.15. Esta última función, llamada *setPmodWifiAddresses()*, perteneciente al fichero System.c y se llama nada más iniciar la aplicación software para que estos drivers conozcan las direcciones del periférico WiFi.

```
void setPmodWifiAddresses( u32 SPI_BASEADDRESS,
                           u32 WFGPIO_BASEADDRESS,
                           u32 WFCS_BASEADDRESS,
                           u32 TIMER_BASEADDRESS )
{
    WF_GPIO_BASEADDRESS  =  WFGPIO_BASEADDRESS;
    WF_SPI_BASEADDRESS   =  SPI_BASEADDRESS;
    WF_CS_BASEADDRESS    =  WFCS_BASEADDRESS;
    WF_TIMER_BASEADDRESS =  TIMER_BASEADDRESS;
}
```

Figura 5.15. Asignación de las direcciones de memoria a las variables de los drivers que las modelan.

5.2.3.3. Drivers MRF24G

A partir de este punto, entran en juego los drivers utilizados para programar el circuito MRF24WG0MA (MRF24G drivers). Estos drivers utilizan funciones comentadas anteriormente, como por ejemplo *WF_SpiInit()*, para poder controlar el periférico WiFi y así programar el chip. En la figura 5.17 se muestra la función de inicialización *WF_Init()*, la cual se encarga de iniciar la secuencia de configuración inicial del circuito MRF24WG0MA.

La función sigue el flujograma observado en la figura 5.16 en la que, como se puede observar, llama en primer lugar, entre otras cosas, a la función de inicialización, *WF_SpiInit(),* del módulo SPI que contiene el periférico y que se explicó en el punto anterior, junto con funciones de inicialización del resto de módulos. También lleva a cabo la secuencia de arranque del circuito MRF24WG0MA, cambiando el valor de nivel alto a bajo de la señal hibernate y cambiando el valor de alto a bajo nivel de la señal de reset, ambas siguiendo un tiempo determinado al realizar dichos cambios de valores en las señales, lo cual es necesario para un correcto arranque del circuito MRF24WG0MA. Por último, resetea el PLL que contiene el circuito y limpia el buffer para que no se encuentren desde el arranque mensajes indeseados provenientes del circuito y asigna un valor a una variable que controla una máquina de estados implementada en otra función encargada de completar el proceso de inicialización del circuito.

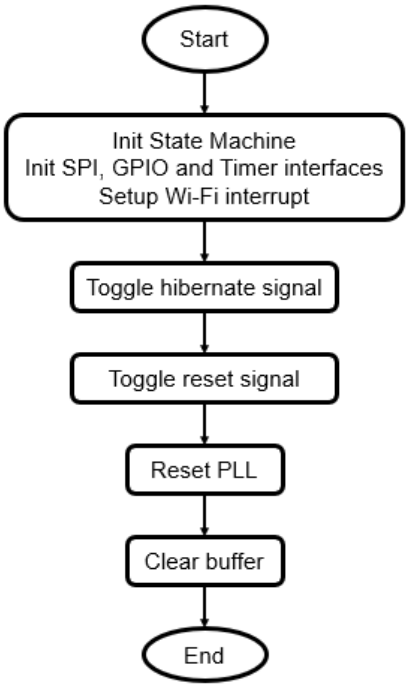


Figura 5.16. Flujograma de la secuencia de inicialización del chip MRF24WG0MA.

```
void WF_Init(void)

{
    uint32_t    tStart = 0;


    UdStateInit();   // initialize internal state machine
    WF_SpiInit();    // initialize the SPI interface
    WF_GpioInit();   // initialize HIBERNATE and RESET I/O lines
    WF_TimerInit();  // initialize and start the 1ms timer
    EventQInit();    // initialize WiFi event queue

    // Take chip out of hibernate and out of reset
    // must be done before calling ResetPll()

    // Toggle the module into and then out of hibernate
    WF_GpioSetHibernate(WF_HIGH);
    tStart = SYSGetMilliSecond();
    while(SYSGetMilliSecond() - tStart <= 2);
    WF_GpioSetHibernate(WF_LOW);
    tStart = SYSGetMilliSecond();
    while(SYSGetMilliSecond() - tStart <= 300);

    // Toggle the module into and out of reset
    WF_GpioSetReset(WF_LOW);
    tStart = SYSGetMilliSecond();
    while(SYSGetMilliSecond() - tStart <= 2);
    WF_GpioSetReset(WF_HIGH);
    tStart = SYSGetMilliSecond();
    while(SYSGetMilliSecond() - tStart <= 5);
```

```
    // MRF24WG silicon work-around
    // needed for A1 silicon to initialize PLL values correctly
    ResetPll();

    // no mgmt response messages received
    ClearMgmtConfirmMsg();

    g_mrf24wgResetState = MRF24WG_RESET_START;

}
```

Figura 5.17. Secuencia de inicialización del chip MRF24WG0MA.

Todo esto se corresponde con los drivers de más bajo nivel del BSP, que se encargan de controlar el chip a través de todas las capas de drivers que se han visto. En este caso se ha observado como para tan solo inicializar el chip, es necesario ir a travesando capas de software hasta llegar a la dirección de memoria necesaria que tiene una relevancia con el hardware diseñado previamente en la FPGA o lógica programable. Aunque se ha visto solo lo relativo

a la inicialización del chip, existen muchas más funciones para controlar el circuito con todo lo necesario para poder utilizar la pila TCP/IP sobre él.

5.2.4. Pila TCP/IP

Todos los drivers explicados anteriormente son utilizados por la pila TCP/IP utilizada, llamada DEIPck (*Digilent Embedded IP STACK*) [15], para poder llevar a cabo, junto con la arquitectura hardware que presenta el sistema, todas las funcionalidades de las capas que posee. Mientras que la capa física es realizada directamente por el circuito MRF24WG0MA, es necesario que el resto de las capas de la pila TCP/IP estén desarrolladas a nivel software en el dispositivo que controle la comunicación, en este caso el SoC Zynq. Estas capas comprenden la capa de enlace, de red de transporte y de aplicación, tal y como se muestra en la figura 5.18, en la que se muestran los distintos protocolos que se pueden implementar en cada capa.

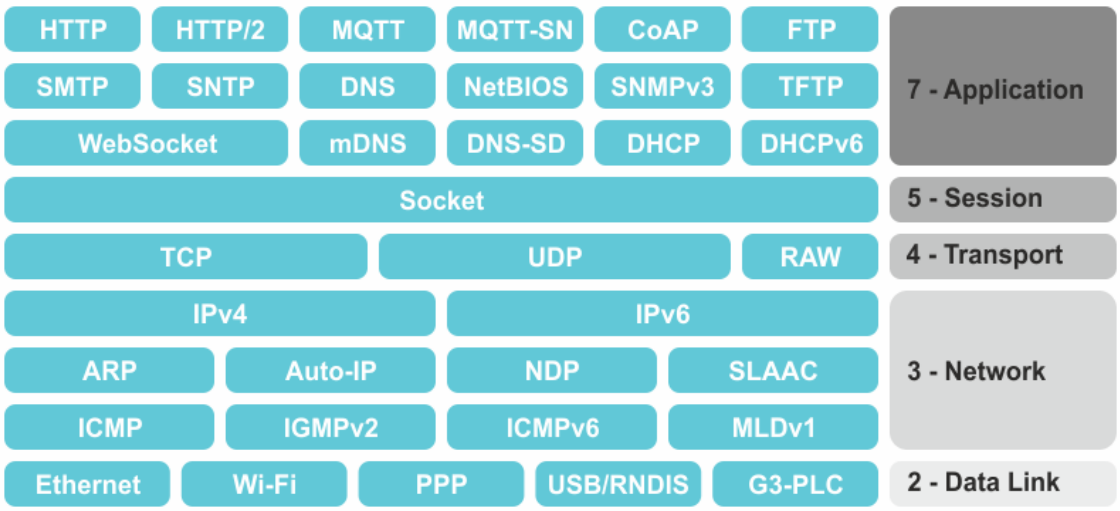


Figura 5.18. Capas de una pila TCP/IP con algunos de los protocolos posibles[20].

5.2.4.1. Capa de Enlace

La capa de enlace es la encargada de transmitir los datos a través del chip MRF24WG0MA, actuando como enlace físico. Para ello es necesario que genere las tramas de datos, aporte una dirección MAC al enlace, gestione los errores que pueden llevarse a cabo y se ocupe del control de flujo. En el caso de este proyecto, se utiliza la familia de normas IEEE 802.11 para realizar todo lo comentado bajo el soporte físico que ofrece el periférico WiFi. La capa de enlace

[20] https://winddoing.github.io/images/2018/08/net_layout.png

presente en la pila DEIPck, se encuentra en el fichero *LinkLayer.c* y en él se observan todas las funciones necesarias para llevar a cabo dicha capa.

5.2.4.2. Capa de Red

La capa de red, también incorporada, tiene como misión que los datos lleguen al destino, aunque ambos puntos, en este caso el módulo WiFi y el PC, no tengan conexión directa. En este caso, normalmente, se utiliza el protocolo de internet o IP, específicamente IPv4. El protocolo IP realiza estas tareas, entre otras cosas, gracias a la dirección IP que incorpora en sus cabeceras, las cuales aportan la dirección de origen y destino para que los enrutadores, o como en este proyecto, el punto de acceso reenvíe los paquetes al dispositivo correcto. La capa de red se encuentra desarrollada en el fichero *InternetLayer.c*, encargándose de realizar todas las tareas descritas anteriormente.

5.2.4.3. Capa de Transporte.

En la capa de transporte, encargada de transportar los datos desde la máquina origen a la de destino de manera fiable, se ha hecho uso del protocolo TCP, aunque la pila también posee desarrollado el protocolo UDP. Se ha usado TCP debido a que este protocolo de transporte garantiza que los datos sean entregados a su destino sin errores y en el mismo orden que salieron, gracias a su mecanismo de transferencia de datos basados en la emisión y recepción de ACKs. Esto mismo no se garantiza con UDP, ya que no tiene confirmaciones basados en estos ACKs, ni control de flujo, lo que puede provocar que los paquetes se adelantes los unos a los otros, y también lleguen de forma incorrecta. Debido a esto se eligió TCP, ya que no es posible que los datos que se reciban para configurar las balizas del sistema ultrasónico tengan errores que podrían modificar notablemente el funcionamiento del sistema. La pila DEIPck proporciona, por tanto, funciones para crear sockets TCP que sean los encargados de transmitir y recibir los datos que se quieren enviar desde el origen al destino en la aplicación que se ha desarrollado. Estas funciones que invocan a la capa de transporte, y a su vez a capas subyacentes, son las que se llamarán desde el programa principal para desarrollar la aplicación correctamente, teniendo en cuenta consideraciones para inicializar la pila DEIPck que se explicarán en la siguiente sección. Los sockets, por lo tanto, sirven de interfaz entre las capas de la pila TCP/IP y la propia aplicación del programa, resultando muy útiles a la hora para el desarrollador que necesite usar la pila DEIPck. Las funciones desarrolladas para implementar la capa de transporte se encuentran en varios ficheros en los que cabe destacar, los ficheros TCP.c, TCPServer.c, TCPSocket.c y TCPSateMachine entre otros.

5.2.5. Aplicación

La aplicación desarrollada, no proporcionada por Digilent Inc., consiste en la creación de un servidor TCP sobre la Zedboard y el módulo WiFi. Una vez creado este servidor WiFi, que ha de estar conectado al router, cualquier PC conectado al mismo router, es decir, que estén en la misma subred, puede conectarse a dicho servidor y mandar los datos oportunos para poder configurar el sistema ultrasónico. Para poder desarrollar esta aplicación es necesario, como se vio en el apartado anterior, conocer cada una de las capas de la pila TCP/IP ofrecida por Digilent Inc. e inicializarlas de la manera adecuada.

5.2.5.1. Consideraciones en el Desarrollo

Para realizar el desarrollo es necesario seguir unas consideraciones de diseño ofrecidas por Digilent para que todo funcione correctamente. En primer lugar, se ha de seguir una estructura de inicialización similar a las dada por la figura 5.19, en la que se muestran las funciones que han de llamarse de la pila para inicializar cada capa de la pila DEIPck.

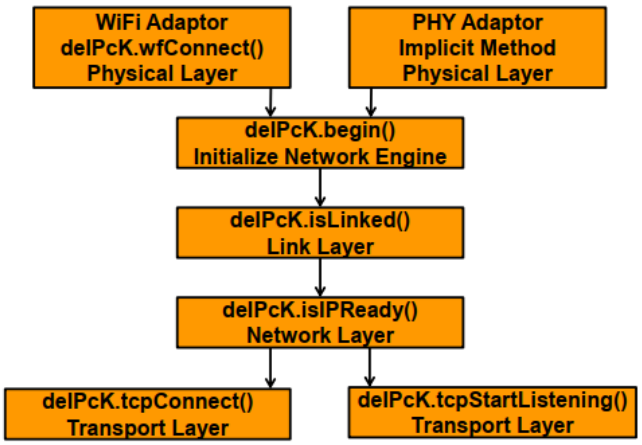


Figura 5.19. Inicialización de cada una de las capas de la pila DEIPck.

Una vez inicializada la pila es necesario seguir el esquema visto en la figura 5.20 sugerido por Digilent, en el que se muestra cómo poder llevar a cabo la programación del servidor. Para el desarrollo se ha seguido un esquema parecido, que consiste básicamente en, como se muestra en la imagen, declarar un objeto de tipo servidor y dejarlo en escucha a través de la función respectiva hasta que un cliente se conecte a él correctamente y se puedan empezar a trasferir los datos de uno a otro. Toda esta secuencia debe, por lo tanto, estar procesándose continuamente para que la conexión entre el módulo WiFi y el PC no desaparezca. Además, la pila necesita llamar a una función periódica que

mantenga la pila activa, lo que hace que la estructura de la aplicación relacionada con la comunicación se base en el bucle observado en el segundo esquema de la figura.

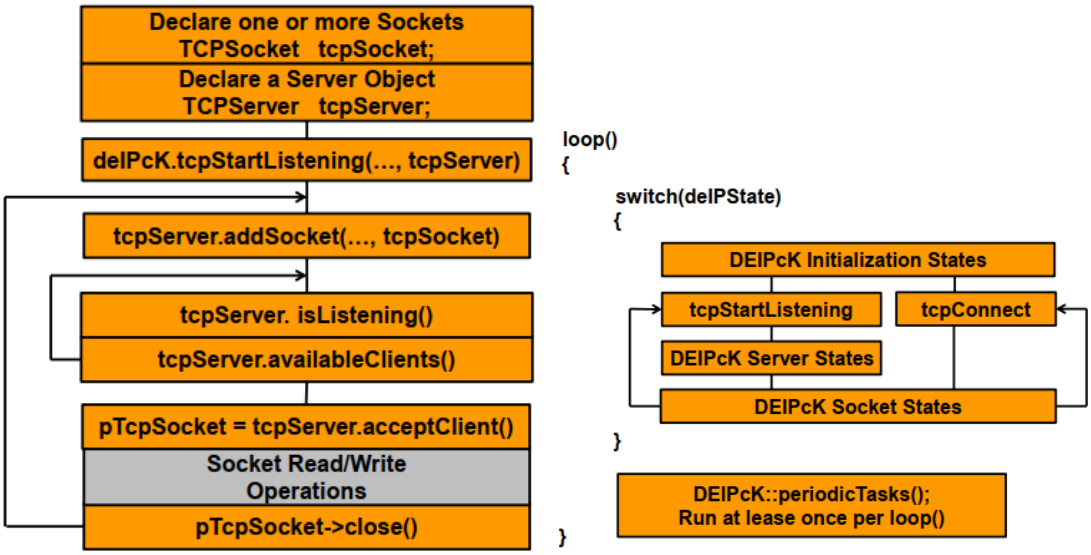


Figura 5.20. Secuencia de funcionamiento de un servidor TCP.

5.2.5.2. Funcionamiento del Servidor TCP Desarrollado

Siguiendo los diagramas de flujo anteriores se ha llevado a cabo una función, que se repite indefinidamente en bucle durante la operación del sistema. Esta función, siguiendo el diagrama de estados dado en la figura 5.21, realiza la conexión del módulo WiFi con el ordenador siguiendo un modelo cliente servidor, en el que el servidor es el sistema ULPS. A continuación, se explicarán cada uno de los estados.

- Ip_Error → Error producido en la pila DEIPck
- Connect → Conexión correcta con el módulo WiFi
- N_sockets → Número de sockets disponibles
- N_Clients → Número de clientes disponibles
- TCP_Client → Conexión correcta con el cliente
- Data → Bytes recibidos
- Timeout → Ciclos de reloj sin recibir ningún mensaje
- End → Finalización de la entrega de mensajes

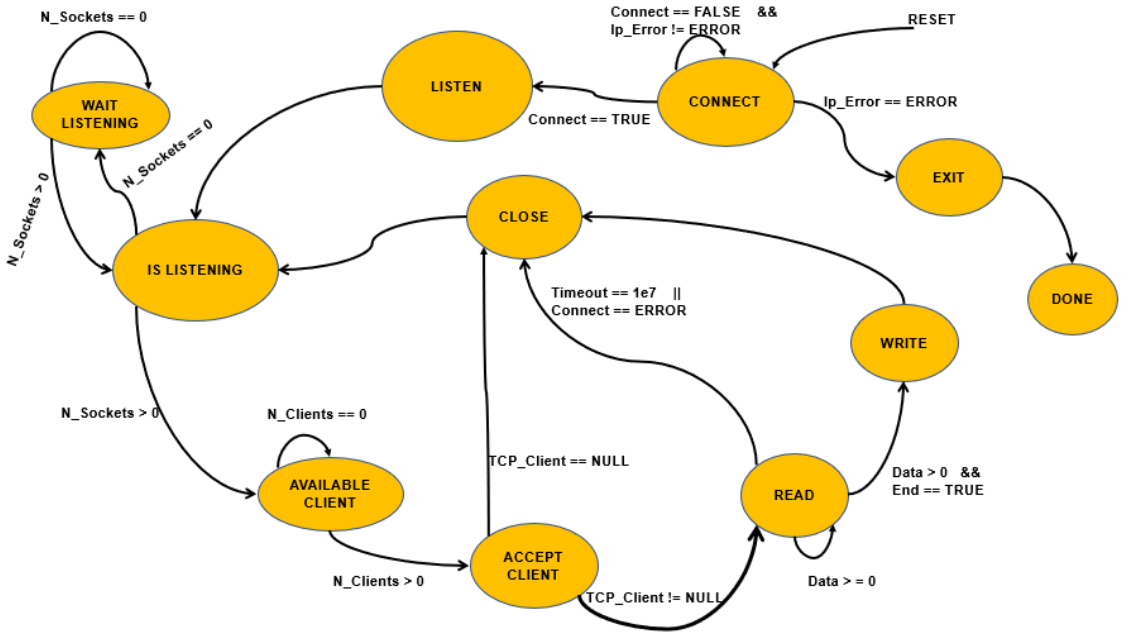


Figura 5.21. Máquina de estados del servidor TCP desarrollado.

- **Estado CONNECT**. Es el primer estado en el que se entra al arrancar el sistema. Al entrar se llama al método *deIPcK.wfConnect()*, al cual es necesario pasarle por parámetros tres elementos de gran importancia a la hora de la conexión: el SSID (*Service Set Identifier*), es el nombre dado al punto de acceso; su contraseña; y una variable de estado. El número de argumentos puede variar dependiendo de la seguridad del router, que, en este caso, tiene un sistema de seguridad WPA2 (*WiFi Protected Access 2*). Si esta operación se realiza correctamente se llama a la función *deIPcK.begin()*, a la que se le pasa como argumento una IP que identifique al servidor, pudiendo ser una IP estática o una IP dinámica que adquirirá del router mediante el protocolo DHCP, introduciendo en la función la IP 0.0.0.0. Después se reinicia la interrupción BCUI, ya que al inicializar el módulo WiFi se deshabilitan todas las interrupciones. Por último, se pasa al estado LISTEN. Si la conexión se realiza de forma incorrecta en el inicio del proceso explicado se pasa al estado de conexión EXIT.

- **Estado Listen.** En este estado se ha de crear el servidor que este en escucha en un determinado puerto, llamando a la función deIPcK.*tcpStartListening()*, a la cual hay que introducir como argumentos el puerto del servidor y una variable de tipo TCPServer, la cual se utilizará para todas las operaciones posteriores. Si la llamada a esta función se realiza sin errores, se inicia la variable socket del tipo TCPSocket con el método *tcpServer.addSocket().* De este estado se pasa directamente al estado ISLISTENING.

- **Estado ISLISTENING.** En este estado, a partir del método *tcpServer.isListening()*, se puede saber si el servidor está esperando una conexión procedente de un cliente de forma. Si es así, adquiere, cuando se le ha asignado de forma dinámica, la dirección IP, y pasa al estado AVAILABLECLIENT. Si no, pasa al estado WAITSLISTENING.

- **Estado WAITSLISTENING.** Aquí, el programa espera hasta que el servidor esté correctamente esperando peticiones y si es así, volver al estado ISLISTENING para poder llegar al estado deseado.

- **Estado AVAILABLECLIENT.** En este estado se espera que un cliente realice una petición de conexión. Esto puede observarse mediante la función *tcpServer.availableClients()* que devuelve el número de clientes que intentan acceder al servidor. Si es así, se pasa al estado ACCEPTCLIENT.

- **Estado ACCEPTCLIENT.** En este estado se acepta la petición por parte de un cliente. Esto se realiza a través del método *tcpServer.acceptClient()* y se comprueba que la conexión es correcta con *ptcpClient->isConnected()*, siendo ptcpClient el objeto del socket del cliente de la clase TCPSocket. Si la operación de aceptación se realiza correctamente, se pasa al estado READ.

- **Estado READ.** Éste es el estado en el que se efectúa la configuración del sistema ultrasónico. Si el socket del cliente tiene en su buffer datos disponibles para la lectura, lo cual se observa a través del método *ptcpClient->available()*, se realiza la lectura de estos datos mediante la función *ptcpClient->readStream()*. Una vez obtenidos los datos, éstos se envían a otra función, llamada *newConfiguration()*, que es la encargada de realizar la nueva configuración del sistema y que sigue un protocolo propio en la recepción del mensaje que se verá en el siguiente punto. Una vez acabada la transferencia de todos los datos de forma correcta, siguiendo el protocolo dado por la anterior función, se pasa al estado WRITE. Si durante la transferencia el cliente cierra la conexión con el servidor, lo cual se puede observar con la función *ptcpClient->isConnected()*, se pasa al estado CLOSE. Cabe destacar que, si no se reciben ningún tipo de dato durante un tiempo establecido, el servidor cierra automáticamente la conexión pasando al estado CLOSE.

- **Estado WRITE.** En este estado, siempre que la conexión entre el cliente y el servidor siga abierta, se realiza el envío de un ACK al cliente para que éste se cerciore de que ha enviado los datos de forma correcta siguiendo la estructura que se comentará en el siguiente punto. Este mensaje de envío contiene únicamente el número de bytes recibidos por parte del servidor, correspondiente con el número de bytes que contienen los códigos recibidos.

- **ESTADO CLOSE.** En este estado se cierra la conexión con el cliente TCP y se vuelve al estado ISLISTENING, por si vuelve a aceptarse otra conexión para volver a configurar el sistema ultrasónico.

- **Estado EXIT.** Si surge algún error relacionado con el funcionamiento de la pila TCP/IP, ya sea algún fallo del propio módulo WiFi o una incorrecta inicialización, se llega a este estado, que cierra el servidor TCP y pasa al estado DONE.

- **Estado DONE.** A este estado se llega únicamente si ha habido algún fallo a través del estado EXIT comentado. Es un estado ocioso, en el que no se realiza ninguna operación y que existe únicamente para mantener el programa activo en un bucle infinito.

5.2.5.3. Recepción de Mensajes

Siguiendo estas consideraciones, una vez que se conecta un cliente al servidor existe un protocolo en la recepción de los datos transmitidos desde la aplicación del PC, que se ha reutilizado del sistema previo comentado en el capítulo 4 del documento y que será explicado a continuación. Este protocolo está realizado en la función newConfiguration(), basada en otra máquina de estados, a la que se le pasa los datos que contiene el buffer del socket y el número de datos recibidos. La primera operación que realiza la función antes de entrar en la máquina de estados es valorar que si el dato recibido, teniendo en cuenta la codificación ASCII, varía entre 48 y 57, con ambos inclusive, corresponderán a un número entero y si son más a una letra. Esta última operación junto la máquina de estados están ubicadas dentro de un bucle que hace sucesivas iteraciones hasta que se han leído todos los datos que contiene el mensaje recibido. Esta secuencia está representada en la figura 5.22 mediante un flujograma.

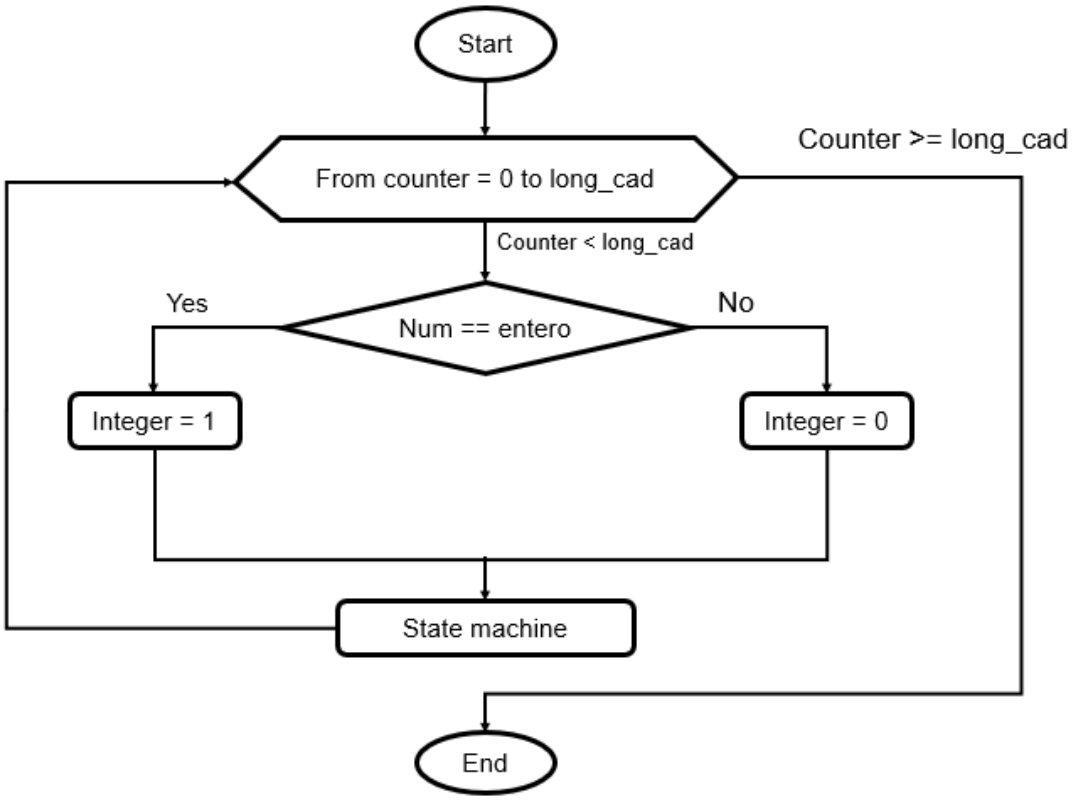


Figura 5.22. Flujograma de la función de recepción de datos.

A continuación, se muestra en la figura 5.23 la máquina de estados implementada para llevar a cabo la recepción de mensajes y poder, de esta manera, conseguir los datos correspondientes a las señales que se desean transmitir por las balizas ultrasónicas, así como otros datos de configuración, como es el periodo de transmisión de dichas señales.

- InData → Datos recibidos existentes en el buffer del socket
- Integer → Indica si el dato recibido es un número entero
- Completed → Indica si el dato recibido está completo
- Enter → Corresponde con el número 32 del código ASCII

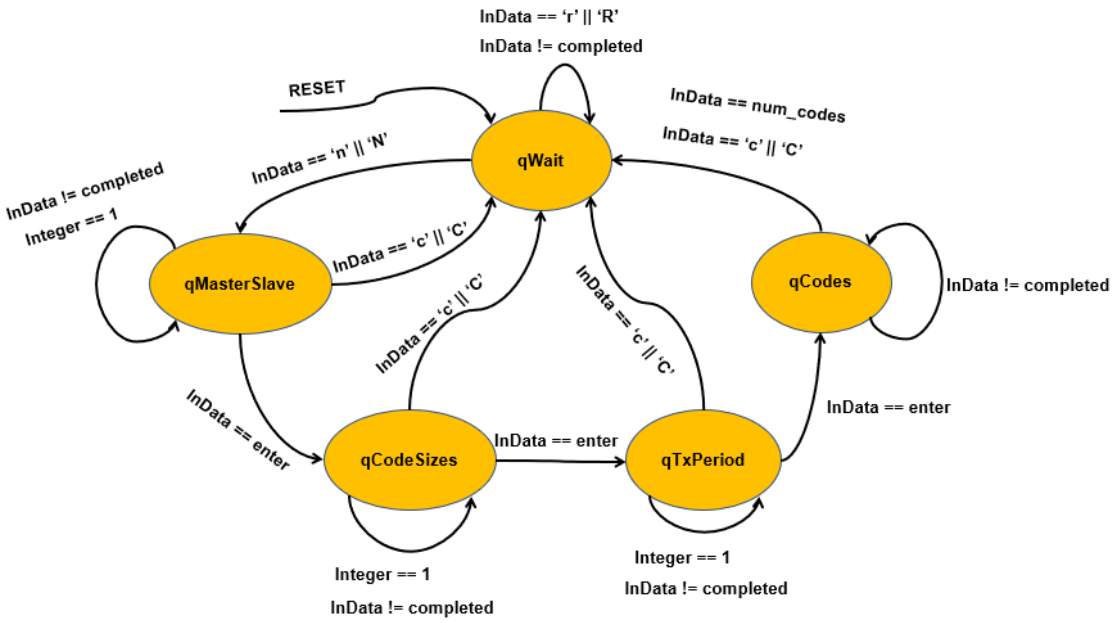


Figura 5.23. Diagrama de estados de los mensajes TCP recibidos.

- **Estado qWait.** Este es el primer estado en el que entra el sistema una vez se reciben datos por primera vez en el estado READ del diagrama de estados anterior. Si se recibe el carácter 'n' o 'N', se inicializan a cero los valores de configuración y las señales guardadas pasando directamente al estado qMasterSlave tras deshabilitar el periférico que desarrolla las funciones de transmisión para que no emita señales en el trascurso de la nueva configuración. Si se recibe una 'r' o 'R', esto querrá decir que la función que se intenta guardar para transmitir no es soportada por el sistema ULPS que se ha desarrollado siguiendo en el mismo estado. Si no se reciben ninguno de estos caracteres se continuará, también, en el estado.

- **Estado qMasterSlave.** Si se recibe una 'c' o 'C' se cancelará la nueva configuración y se volverá al estado qWait. Si en cambio, se recibe un número entero, corresponderá con la asignación del sistema como esclavo o como maestro y éste se guardará en la variable destinada a ello Si en la siguiente iteración del bucle se recibe un enter, se pasará al estado qCodesSizes.

- **Estado qCodeSizes.** Este estado tiene un funcionamiento como el anterior, con la diferencia de que el valor numérico corresponde con el tamaño del código que se desea transmitir. Al igual, si se recibe el carácter que representa un enter, se pasa al siguiente estado, qTxPeriod. Esta variable puede aumentar también con siempre en cada incremento del bucle se reciba un entero.

- **Estado qTxPeriod.** El estado es también igual que los anteriores a diferencia de que se recibe el periodo de transmisión. Esta variable puede aumentar también en cada incremento del bucle.

- **Estado qCodes.** En este estado se reciben los códigos de las señales que se desean guardar para transmitir. Escribiendo dichos datos en las memorias dual-port del periférico o en la memoria DDR3 externa al SoC dependiendo de si la longitud de los códigos supera los 16 k. También se guardan los códigos en la memoria flash para que al arrancar el sistema se tenga una copia de estos códigos. Por último, una vez que se alcanza el tamaño de los códigos esperados, se vuelve al estado qWait y se asigna un uno a una variable encargada de que la máquina de estados correspondiente al servidor TCP pase al estado WRITE para enviar el ACK a la aplicación del PC externo y así completar el protocolo a seguir.

5.2.5.3. Funcionamiento Global del Sistema

El funcionamiento global del sistema debe ser exactamente igual que el que había sido desarrollado en el capítulo 3 del presente documento cumpliendo los mismos requisitos, que los mostrados en el punto 4.1. La única diferencia, como ya se ha comentado, ha sido la incorporación del módulo WiFi, mediante sus distintas fases de implementación, en el sistema ULPS, permitiendo de esta forma mejorar notablemente su arquitectura. Aun así, este cambio, debe ser opaco para el usuario del sistema, que lo único que ha de tener en cuenta es cerciorarse de que exista una buena comunicación inalámbrica entre el PC externo, el punto de acceso y el módulo WiFi una vez que el sistema está probado en el entorno donde se quiere instalar y posteriormente grabado en la plataforma de desarrollo, en este caso, la Zedboard.

Por último, en el Anexo se encuentra el código C de toda la aplicación desarrollada junto con el código reutilizado que ya había sido implementado en el sistema previo, siendo éste ordenado en distintos ficheros según su finalidad dentro del proyecto. Todo este código está dentro del proyecto BCU_top generado tal y como se explicó en el punto 5.2.1.

Capítulo 6

Resultados Experimentales

En este capítulo se mostrarán los resultados experimentales conseguidos al haber implementado el sistema ultrasónico, que ha sido previamente configurado. También se observarán los datos de ocupación de la FPGA que han sido necesarios para la implementación del sistema en el Zynq.

6.1. Pruebas Reales

Una vez ejecutado el sistema completo, caben destacar dos fases en la propia ejecución. Una relativa a la conexión cliente-servidor entre el PC externo y el módulo WiFi, donde se configura el sistema. Y otra, relativa a que el sistema sea capaz, tal y como cumplía el sistema previo a éste, de, a partir de la configuración grabada en memoria flash, transmitir las señales por las balizas con distintas cantidades de muestras.

6.1.1. Pruebas Conexión

Una vez que se inicia la aplicación, tal y como se explicó en el capítulo anterior, el bucle infinito donde corre la aplicación TCP desarrollada intenta crear una conexión con un cliente. Para ello, el cliente debe conectarse a la dirección TCP asignada, ya sea fija o dinámica. En la figura 6.1 se muestran las trazas de depuración generadas a través de un puerto serie, en el que se muestra que el proceso de inicialización del servidor TCP, se ha generado con éxito, esperando a que un cliente se conecte a él.

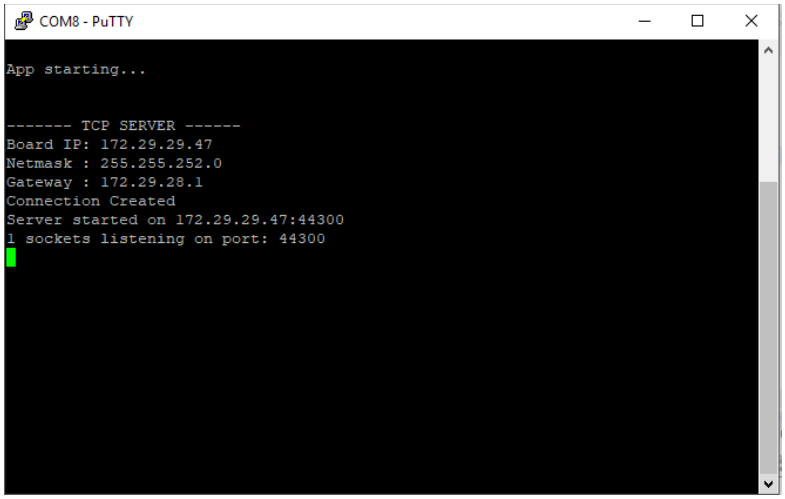


Figura 6.1. Trazas de depuración obtenidas por terminal serie durante la creación del servidor.

Cuando el cliente TCP, generado en la aplicación del PC externo, se conecta a la dirección y puerto asociados al servidor, empieza la transferencia de datos que configuran el sistema. En primer lugar, se transmiten los datos relativos a la configuración, tales como, el periodo de transmisión de las señales, la configuración del sistema como maestro o servidor y la longitud de las muestras, para que éstas se guarden posteriormente en la memoria adecuada. Después, se transmiten los datos relativos a las muestras digitales de cada señal a transmitir por el sistema ultrasónico. Y, una vez que esto se ha transferido, el servidor envía los ACKs para que el cliente se cerciore de que se ha recibido todo correctamente y cierre la conexión. Estos dos últimos pasos se muestran en las figuras 6.2 y 6.3 respectivamente, donde se observan los bytes recibidos en cada transferencia de datos TCP y el ACK enviado que hace que la conexión se cierre.

Figura 6.2. Trazas de depuración obtenidas por terminal serie durante la transferencia de datos.

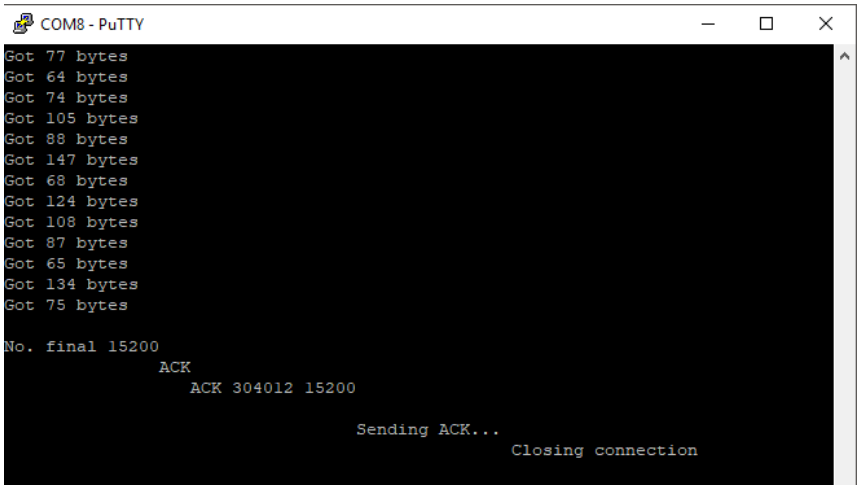


Figura 6.3. Trazas de depuración obtenidas por terminal serie durante el final de la conexión.

6.1.2. Pruebas del Sistema Completo

Cuando el sistema ha sido correctamente configurado, comienza la transmisión de señales ultrasónicas desde las balizas. Para mostrar estos resultados se han realizado capturas con un osciloscopio digital capaz de visualizar las señales en dos de las salidas de los DACs que forman el sistema, el resto de las medidas que se pueden tomar del resto de DACs son muy similares y por eso no han sido mostradas a continuación.

En la figura 6.4 se observan las dos salidas de los DACs, en la que se ha configurado una modulación BPSK (*Binary Phase Shift Keying*) con una portadora de 40kHz, codificadas con secuencias Kasami de 1023 bits. En ella se muestra cómo se sincronizan los dos canales de transmisión, enviando la señal cuando la última ha dejado de hacerlo. Para el resto de los canales se realiza exactamente lo mismo, haciendo que durante todo el tiempo se transmitan las señales por las distintas balizas.

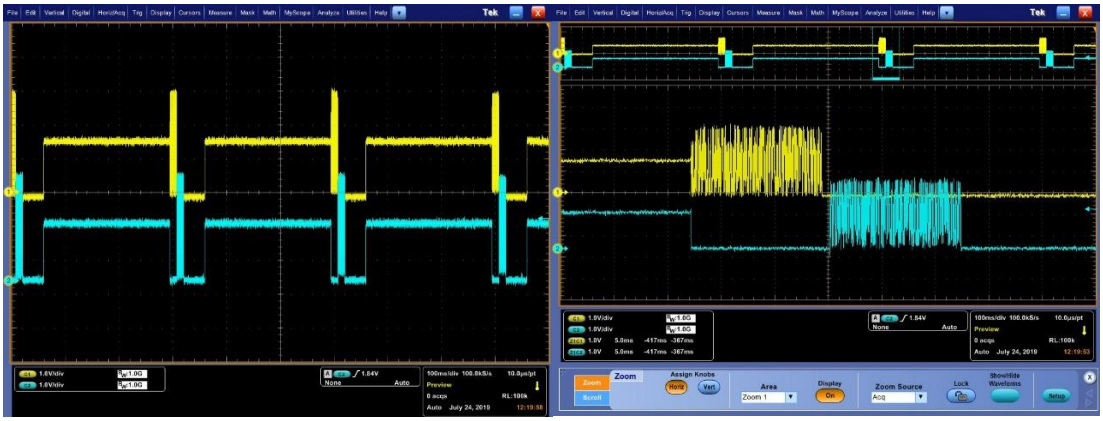


Figura 6.4. Transmisión de una modulación BPSK de 40kHz y secuencias Kasami de 1023 bits en dos de las salidas de los DACs.

En la figura 6.5 se muestra la misma modulación BPSK con secuencias Kasami, en la que en la primera imagen se ha configurado con un periodo de transmisión de 200ms y en el segundo de 300ms, apreciando dicha diferencia de periodo.

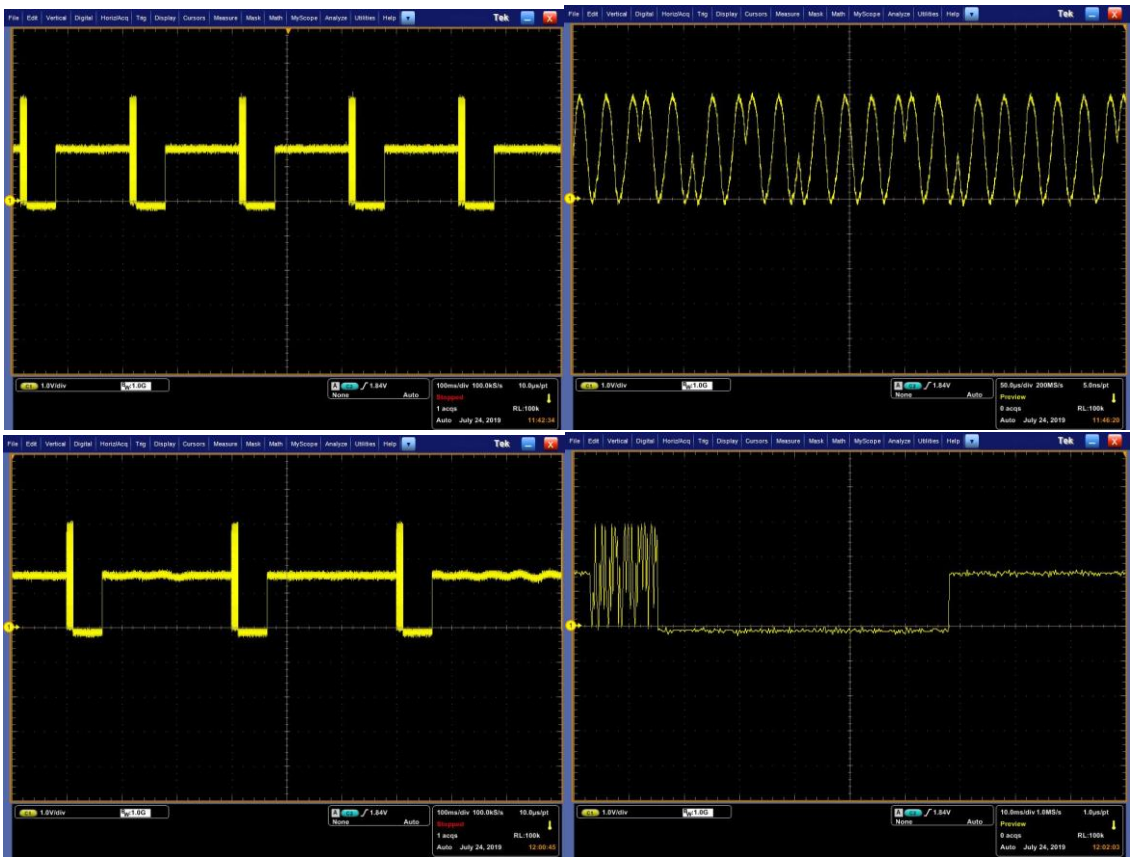


Figura 6.5. Modulación Kasami con distintos periodos de trasmisión. Arriba con 200ms y abajo 300ms.

En la figura 6.6 se muestra dos señales sinusoidales que poseen distintos números de muestras por periodo. La primera con 30k y la segunda con 120k.

Esto provoca que la señal con un menor número de muestras sea peor, haciendo que el DAC no pueda reconstruir la señal sinusoidal de la mejor forma posible.

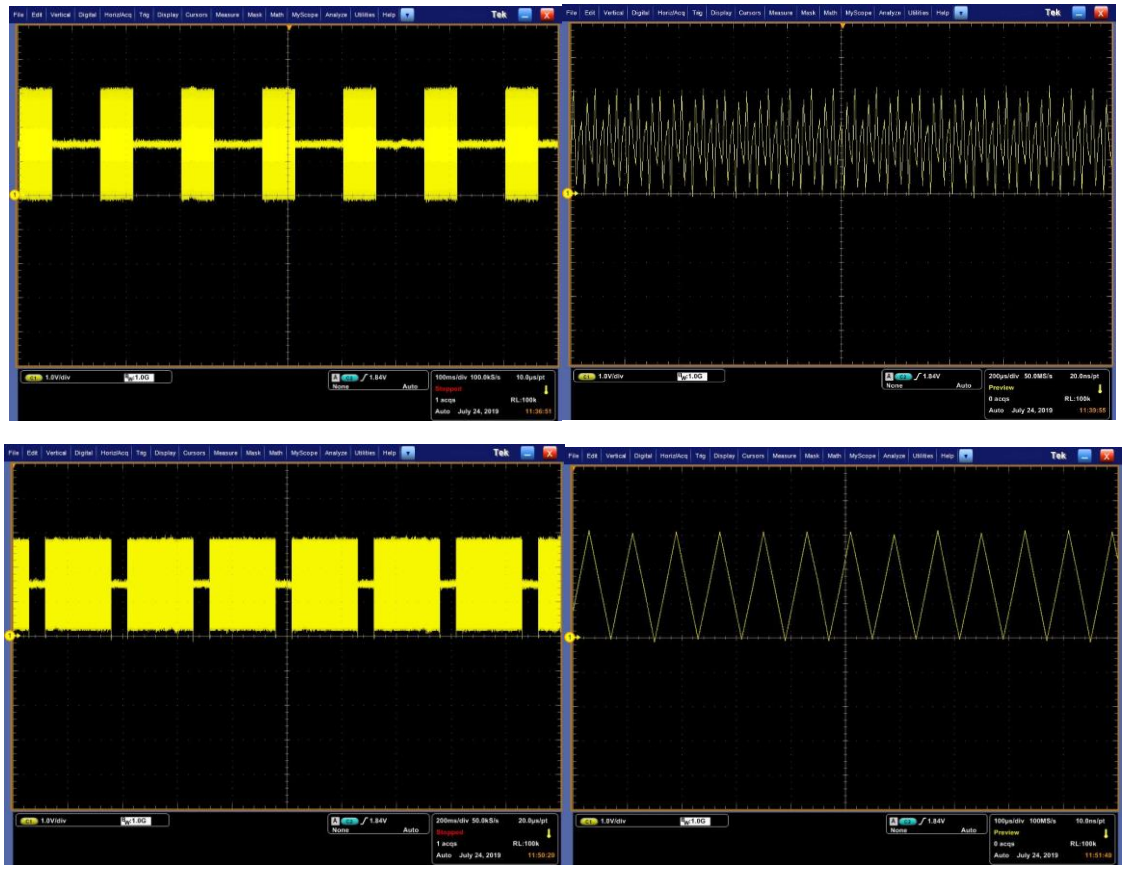


Figura 6.6. Modulación sinusoidal con distintas muestras. Arriba con 60k y abajo con 120k.

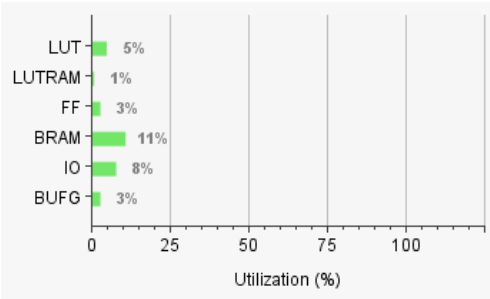
Por último, se muestra en la figura 6.7 una modulación BPSK con una secuencia Zadoff-Chu con un periodo de 300ms con 2560 muestras por periodo.

Figura 6.7. Modulación BPKS y secuencia Zadoff-Chu con un periodo de 300ms y 2560 muestras.

6.2. Datos de Ocupación de la FPGA

Los recursos utilizados por parte de la FPGA se han obtenido de la herramienta de Xilinx utilizada en todo el proyecto, Vivado. Éstos se pueden extraer del proceso de síntesis como en el de implementación, siendo en el último donde se obtiene una mayor precisión, ya que durante la implementación hay una fase de optimización al realizar el ruteado de la FPGA.

Por otro lado, en la figura 6.8 se muestran los datos finales que se han utilizado después de la implementación, también en forma de gráfica y tabla, pudiéndose observar el porcentaje de recursos estimados.



Resource	Utilization	Available	Utilization %
LUT	2457	53200	4.62
LUTRAM	155	17400	0.89
FF	3134	106400	2.95
BRAM	16	140	11.43
IO	15	200	7.50
BUFG	1	32	3.13

Figura 6.8. Recursos utilizados durante la implementación.

Capítulo 7

Conclusión

En este capítulo se analizará el trabajo realizado, comentando los resultados obtenidos en su desarrollo. Se tratarán los problemas encontrados y los logros obtenidos en su elaboración, así como posibles líneas futuras de desarrollo que permitan dotar al sistema de mejoras.

7.1. Conclusiones

El objetivo de este trabajo ha sido continuar el desarrollo del sistema ULPS desarrollado por el grupo de investigación GEINTRA [1] y dotarlo de una conexión inalámbrica con un ordenador externo que estuviera ejecutando un programa cliente. Para ello, ha sido necesario incorporar al sistema, basado en un SoC un módulo WiFi de bajo coste [13], siendo necesario tanto un desarrollo hardware como software para su implementación.

El trabajo ha podido realizarse en la fase de desarrollo tanto hardware como software debido a la multitud de información que proporciona el fabricante del módulo WiFi [16], tanto para la creación del módulo hardware generado en Vivado, como el uso de la pila TCP/IP en la cual se basa el servidor TCP creado [15]. Aun con las facilidades proporcionada por el fabricante fue necesario realizar un estudio previo del chip al completo para comprender su funcionamiento.

El estudio previo, en primer lugar, se fijó en la función del módulo IP Core que debía incorporarse al diagrama de bloques completo del sistema hardware de la FPGA a través de Vivado. Una vez conocida todos los elementos que lo componían, se conectó de forma correcta, a través de conexiones AXI que accedían a los módulos del IP Core, con el sistema de procesamiento del SoC.

Una vez generado el diagrama de bloques en el hardware ubicado en la FPGA del SoC, se tuvo que estudiar la pila TCP/IP que se iba a incorporar en los drivers del software del sistema. Ésta, una vez añadida junto con drivers de más bajo

nivel que accedían al módulo WiFi, se utilizó para la creación del servidor TCP que debe conectarse con el cliente TCP generado en un PC externo a través del propio módulo WiFi.

La creación del servidor TCP tuvo problemáticas debido a la capacidad del número de datos que permitía obtener el buffer de recepción de mensajes TCP por parte de la pila de protocolos. Pero con un debido procesamiento de estos mensajes se alcanzaron los objetivos.

Por último, el sistema fue validado cambiando la configuración del sistema, creando correctamente la conexión cliente-servidor entre los dos elementos y probando así una cantidad de modulaciones diferentes, viéndose visualizadas en un osciloscopio.

7.2. Trabajos Futuros

En este punto se valorarán ciertos trabajos futuros que pueden seguir en la línea de la mejora de este sistema ultrasónico con conexión WiFi desarrollado en este Trabajo Fin de Grado, mostrándose a continuación algunas de ellas.

- Diseño de un PCB (*Printed Circuit Board)* que incluya el SoC y los periféricos utilizado, incluyendo el periférico WiFi, haciendo el sistema físicamente más pequeño y evitando el uso de la Zedboard, que contiene muchos elementos no utilizados.

- Incorporar un panel solar al sistema ULPS que cargue las baterías para que pueda ser autosuficiente eléctricamente y no haya que preocuparse de su alimentación.

- Incorporar una comunicación WiFi entre las distintas balizas que pueden contener un sistema completo, para no una vez que se configure una baliza se puedan configurar todas sucesivamente y de forma autónoma.

Bibliografía

[1] A Hernández, E Garcia, D Gualda, J.M. Villadangos, F Nombela, J Ureñá, "*FPGA-Based Architecture for Managing Ultrasonic Beacons in a Local Positioning System*", IEEE transactions on instrumentation and measurement, vol. 66, no. 8, August 2017, pp. 1954-1964.

[2] GEINTRA, "*Mejora y robustecimiento de sistemas de localización en interiores para aplicaciones en robótica y asistencia a personas*", [Online]. Available: http://www.geintra-uah.org/projects/mejora-y-robustecimiento-sistemas-localizacion-interiores-aplicaciones-robotica-y-asistenci, último acceso realizado el 21 de marzo de 2020.

[3] Oficina de Coordinación Nacional de Posicionamiento, Navegación, y Cronometría por Satélite de los EEUU, "*Sistema de Posicionamiento Global*", [Online]. Available: https://www.gps.gov/spanish.php, último acceso realizado el 21 de marzo de 2020.

[4] Jordi Salazar, "*Procesadores digitales de señal (DSP) Arquitecturas y criterios de selección*", [Online]. Available: https://studylib.es/doc/4712889/procesadores-digitales-de-se%C3%B1al--dsp-, último acceso realizado el 21 de marzo de 2020.

[5] EL-PRO-CUS, "*Introduction to Application Specific Integrated Circuit (ASIC)*", [Online]. Available: https://www.elprocus.com/application-specific-integrated-circuits/, último acceso realizado el 21 de marzo de 2020.

[6] Digilent, "*ZedBoard (Zynq Evaluation and Development) Hardware User's Guide*", 2012 [Online]. Available: https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf?_ga=2.128929674.788221263.1564503778-104097335.1563878083, último acceso realizado el 21 de marzo de 2020.

[7] L. H. Crockett, R. Elliot, M. Enderwitz "*The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*". Strathclyde Academic Media. 2014.

[8] J. Ureña et al., "*Technical description of locate-US: An ultrasonic local positioning system based on encoded beacons*," in Proc. Int. Conf. Indoor Positioning Indoor Navigat. (IPIN), Oct. 2016, pp. 1–4.

[9] D. Gualda, "*Simultaneous Calibration of Ultrasonic Local Positioning Systems and Mobile Robot Navigation*", Ph.D., Universidad de Alcalá, 2016.

[10] F. Pérez Fermoselle, "*Diseño e Implementación de un SoC para el Control de un LPS Extenso*", TFM, Universidad de Alcalá, 2014, [Online]. Available: https://ebuah.uah.es/dspace/bitstream/handle/10017/21316/TFM-P%c3%a9rez-Fermoselle-2014.pdf?sequence=4&isAllowed=y, último acceso realizado el 21 de marzo de 2020.

[11] M. Maxfield, "*ASIC, ASSP, SoC, FPGA – What's the Difference?*", 2014, [Online]. Available: https://www.eetimes.com/asic-assp-soc-fpga-whats-the-difference/#, último acceso realizado el 21 de marzo de 2020.

[12] Xilinx, "*Zynq-7000 SoC Product Advantages*", [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, último acceso realizado el 21 de marzo de 2020.

[13] Digilent, "*Pmod WiFi Reference Manual*", 2016, [Online]. Available: https://reference.digilentinc.com/_media/reference/pmod/pmodwifi/pmodwifi_rm.pdf, último acceso realizado el 21 de marzo de 2020.

[14] Microchip, "*MRF24WG0MA/MB Data Sheet*", 2012, [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/70686B.pdf, último acceso realizado el 21 de marzo de 2020.

[15] Microchip "Easy Wireless Networking Using the Arduino Compatible chipKIT Platform", 2014, [Online]. Available: https://reference.digilentinc.com/_media/reference/pmod/pmodwifi/networkslides.pdf, último acceso realizado el 21 de marzo de 2020.

[16] Digilent, "Vivado Library, PmodWIFI IP", [Online]. Available: https://github.com/Digilent/vivado-library/tree/master/ip/Pmods/PmodWIFI_v1_0, último acceso realizado el 21 de marzo de 2020.

[17] Digilent, "*PmodDA2 Reference Manual*", 2016, [Online]. Available: https://reference.digilentinc.com/_media/reference/pmod/pmodda2/pmodda2_rm.pdf, último acceso realizado el 21 de marzo de 2020.

Parte III

Recursos y Anexos

Apéndice A

Herramientas y Recursos

En este apéndice se tratarán las herramientas que han sido necesarias para realizar el proyecto, así como los recursos económicos que han sido necesarios.

A.1. Pliego de Condiciones

A continuación, se mostrarán los elementos hardware y las herramientas software utilizadas durante el desarrollo de este proyecto.

A.1.1. Recursos Hardware

- Ordenador portátil ASUS K555L
- Tarjeta de desarrollo ZedBoard
- Módulo Pmod WiFi: WiFi Interface 802.11g
- Módulo Pmod DA2: Two 12-bit D/A Outputs
- Router WiFi
- Cables: USB a USB-micro

A.1.2. Recursos Software

- Sistema operativo: Windows 10 de 64 bits
- Xilinx Vivado Design Suite 2017.4
  - Vivado 2017.4
  - Xilinx SDK 2017.4
- Depurador vía comunicación serie: PuTTY
- Depuración mensajes TCP/IP: Wireshark
- Suite Microsoft Office

- Cliente TCP: Matlab R2018b

A.2. Presupuesto

En este apartado se tratarán los costes relativos al proyecto, desglosando en los diferentes recursos mostrados a continuación.

A.2.1. Recursos Hardware

En la tabla A.1 se muestra el coste relativo a los recursos hardware o físicos necesarios para la realización del trabajo.

Material	***Precio Unidad***	***Unidades***	***Precio Total***	***Amortización***	***Tiempo de Uso***	***Subtotal***
Ordenador	600 €	1	600 €	5 años	6 meses	60 €
Zedboard	485 €	1	485 €	-	-	485 €
***Módulo* WiFi**	23 €	1	23 €	-	-	23 €
Router WiFi	16 €	1	16 €	-	-	16 €
***Módulo* DA2**	20 €	3	60 €	-	-	60 €
Total						***644 €***

Tabla A.1. Coste relativo a los recursos hardware.

A.2.2. Recursos Software

En este apartado se desglosan en la tabla A.2 los costes relativos a los programas informáticos necesarios en el desarrollo del proyecto. Como se pueden observar han tenido un coste cero, ya que en todos se cuenta con licencia de estudiante disponible por la Universidad de Alcalá.

Software	Precio Total	Amortización	Tiempo de Uso	Subtotal
Xilinx Vivado Design Suit	0 €	-	6 meses	0 €
Paquete Office	0 €	-	6 meses	0 €
Matlab R2018B	0 €	-	6 meses	0 €
Total				**0 €**

Tabla A.2. Coste relativo a los recursos software.

A.2.3. Mano de Obra

En la tabla A.3, se observan los costes relacionados con la mano de obra necesaria para realizar el proyecto, fijando un salario medio para un ingeniero graduado, tanto en el diseño del sistema como la redacción de la documentación técnica presente en este libro.

Tarea	Salario (Euros/Hora)	Horas	Subtotal
Diseño del sistema	25 €/h	350	8.750 €
Redacción del libro	15 €/h	150	2.250 €
Total			**11.000 €**

Tabla A.3. Coste relativo a la mano de obra.

A.2.4. Coste total

El coste total de todo el proyecto puede verse resumido en la tabla A.4.

Concepto	Total
Recursos Hardware	644 €
Recursos Software	0 €
Mano de Obra	11.000 €
Coste total	**11.654 €**
IVA (21%)	2.447 €
Importe total	**14.101 €**

Tabla A.4. Coste total del proyecto.

Apéndice B

Anexo

En este anexo se muestra el fichero BCU_top.c donde se ha desarrollado la mayoría del código C explicado durante el documento.

```
/////////////////////////////////////////////////////////////////////////////
///////////
////////////////////////////////// FILE INCLUDES
//////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
///////////
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "math.h"
#include "BCU_top.h"

#include "xil_cache.h"
#include "xbasic_types.h"
#include <xil_types.h>
#include "Xil_exception.h"
#include "platform_config.h"

/*M□o Wifi*/
#include "PmodWIFI.h"
#include "WifiConfig.h"

// Flash memory
#include "xqspips.h"        /* QSPI device driver */

// Peripheral
#include "beaconctrl_top.h"

#include "xgpio.h"



/////////////////////////////////////////////////////////////////////////////
///////////
// Global variables
/////////////////////////////////////////////////////////////////////////////
///////////
// XFlash Instance
static XQspiPs QspiInstance;
u8 ReadBuffer[FLASH_MAX + DATA_OFFSET + DUMMY_SIZE];
u8 WriteBuffer[PAGE_SIZE + DATA_OFFSET];
```

```
Xuint32 param[NUM_WR_REG];
u8 data_flash[FLASH_MAX];
Xuint32 idxFlash, j;
Xuint32 code_data[2*NUM_BEACONS];
int flash_error= 0;

// Peripheral
Xuint32 *per_pnt= (Xuint32*) PERIPHERAL_ADDR;
int code_size, tx_period;
int ext_mode;
u8 peri_control;
int int_count=0;


int code_cnt, rt_code_cnt;
u8 integer=0, state=0, beacon;


// DMA
static Xuint32 b1_ddr_code[C_DMA_SIZE];
static Xuint32 b2_ddr_code[C_DMA_SIZE];
static Xuint32 b3_ddr_code[C_DMA_SIZE];
static Xuint32 b4_ddr_code[C_DMA_SIZE];
static Xuint32 b5_ddr_code[C_DMA_SIZE];


/*WIFI*/
STATE estado_conexion;
TCPServer tcpServer;
TCPSocket rgTcpClient[cTcpClients];
TCPSocket *ptcpClient = NULL;
IPSTATUS status;


int main(void)
{
    /* The MAC address of the board. This should be unique per board */
    //unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01,
0x02 };

    init_platform ();
    xil_printf ("\nApp starting...\n");

    PmodWIFI_Initialize();

    print_app_header ();
    print_ip_settings (&ipMyStatic, &subnetMask, &ipGateway);

    /*Iniciamos la flash y recojemos la ltima confiduraci□el sistema*/
    initFlash ();
    startSystemFromFlash ();

    // now enable interrupts
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

    StartWIFI_Aplication ();

    // Never reached
    cleanup_platform ();
    Xil_DCacheDisable ();
    Xil_ICacheDisable ();

    return 0;
```

```
}



/////////////////////////////////////////////////////////////////////////////
//////////
// Flash functions
/////////////////////////////////////////////////////////////////////////////
//////////
int initFlash (void)
{
    int Status;
    XQspiPs_Config *QspiConfig;


    flash_error= 0;

    // Initialize the QSPI driver so that it's ready to use
    QspiConfig = XQspiPs_LookupConfig(QSPI_DEVICE_ID);
    if (NULL == QspiConfig)
    {
        flash_error= 1;
        return XST_FAILURE;
    }

    Status = XQspiPs_CfgInitialize(&QspiInstance, QspiConfig,
                    QspiConfig->BaseAddress);
    if (Status != XST_SUCCESS)
    {
        flash_error= 1;
        return XST_FAILURE;
    }

    // Perform a self-test to check hardware build
    Status = XQspiPs_SelfTest(&QspiInstance);
    if (Status != XST_SUCCESS)
    {
        flash_error= 1;
        return XST_FAILURE;
    }

    // Set Manual Start and Manual Chip select options and drive HOLD_B pin
high.
    XQspiPs_SetOptions(&QspiInstance, XQSPIPS_MANUAL_START_OPTION    |
                                      XQSPIPS_FORCE_SSELECT_OPTION  |
                                      XQSPIPS_HOLD_B_DRIVE_OPTION);

    // Set the prescaler for QSPI clock
    XQspiPs_SetClkPrescaler(&QspiInstance, XQSPIPS_CLK_PRESCALE_8);

    // Assert the FLASH chip select.
    XQspiPs_SetSlaveSelect(&QspiInstance);


    return XST_SUCCESS;
}


void startSystemFromFlash (void)
{
```

```
    int i;
    Xuint32 read_size;


    readParamFromFlash ();
    // writeRegPERI ((param[0]&0xFFFFFFFE), param[1], param[2]); // Not
enable

    peri_control= param[0];
    code_size= param[1];
    tx_period= param[2];
    rt_code_cnt= 0;

    // AHA1
    /* xil_printf ("\nPER control: %d", peri_control);
    xil_printf ("\nPER size: %d", code_size);
    xil_printf ("\nPER period: %d", tx_period); */
    // return;

    // Limits for first running from flash with no initial data
    // if (peri_control>3 || code_size>20480 || tx_period>30)
    // if (peri_control>15 || code_size>131072 || tx_period>70)
    if (peri_control>15 || code_size>131072 || tx_period>5000)
    {
        peri_control= 0;
        code_size= 0;
        tx_period= 0;
    }

    if (code_size>SIZE_PER_BEACON)
        ext_mode= 1;
    else
        ext_mode= 0;

    writeRegPERI ((param[0]&0xFFFFFFFE), code_size, param[2]); // Not enable

    read_size= NUM_BEACONS*BYTES_PER_SAMPLE*code_size;

    memset(ReadBuffer, 0x00, sizeof(ReadBuffer));
    FlashRead(&QspiInstance, (START_ADDRESS+PARAM_SIZE), read_size,
READ_CMD);

    idxFlash= DATA_OFFSET;

    /* for (i=2048; i<4095; i++)
    {
        xil_printf ("\nNo. %d\t Data %x", i, ReadBuffer[i]);
    } */

    for (i=0;i<code_size/2;i++)
    {
        code_data[0]= (ReadBuffer[idxFlash] + (ReadBuffer[idxFlash+1]*256));
        code_data[5]=  (ReadBuffer[idxFlash+10]) +
(ReadBuffer[idxFlash+11]*256);
        code_data[1]= (ReadBuffer[idxFlash+2] +
(ReadBuffer[idxFlash+3]*256));
        code_data[6]=  (ReadBuffer[idxFlash+12]) +
(ReadBuffer[idxFlash+13]*256);
        code_data[2]= (ReadBuffer[idxFlash+4] +
(ReadBuffer[idxFlash+5]*256));
        code_data[7]=  (ReadBuffer[idxFlash+14]) +
(ReadBuffer[idxFlash+15]*256);
```

```
        code_data[3]= (ReadBuffer[idxFlash+6] +
(ReadBuffer[idxFlash+7]*256));
        code_data[8]=  (ReadBuffer[idxFlash+16]) +
(ReadBuffer[idxFlash+17]*256);
        code_data[4]= (ReadBuffer[idxFlash+8] +
(ReadBuffer[idxFlash+9]*256));
        code_data[9]=  (ReadBuffer[idxFlash+18]) +
(ReadBuffer[idxFlash+19]*256);
        idxFlash+= 20;

        if (ext_mode==0)
            writeDataPERI (i, code_data);
        else
        {
            writeDataDDR (i, code_data);
            if (i<SIZE_PER_BEACON/2)
                writeDataPERI (i, code_data);
            else
                rt_code_cnt= SIZE_PER_BEACON/2;
        }
    }

    writeRegPERI (param[0], param[1], param[2]);    // Enable system
}


Xuint32* readParamFromFlash (void)
{
    memset(ReadBuffer, 0x00, sizeof(ReadBuffer));
    FlashRead(&QspiInstance, (START_ADDRESS), PARAM_SIZE, READ_CMD);
    // Status= XFlash_Read (&FlashInstance, (START_ADDRESS), PARAM_SIZE,
ReadBuffer);

    param[0]= (ReadBuffer[DATA_OFFSET+0] + (ReadBuffer[DATA_OFFSET+1]*256)
            + (ReadBuffer[DATA_OFFSET+2]*(65536)) +
(ReadBuffer[DATA_OFFSET+3]*(16777216)));    // mode master/slave
    param[1]= (ReadBuffer[DATA_OFFSET+4] + (ReadBuffer[DATA_OFFSET+5]*256)
            + (ReadBuffer[DATA_OFFSET+6]*(65536)) +
(ReadBuffer[DATA_OFFSET+7]*(16777216)));    // code_size
    param[2]= (ReadBuffer[DATA_OFFSET+8] + (ReadBuffer[DATA_OFFSET+9]*256)
            + (ReadBuffer[DATA_OFFSET+10]*(65536)) +
(ReadBuffer[DATA_OFFSET+11]*(16777216)));  // tx_period

    return param;
}


void writeFlash (Xuint32 peri_control, Xuint32 code_size, Xuint32 tx_period)
{
    Xuint32 write_size;
    int page_cnt, no_page;
    int i;


    FlashErase(&QspiInstance, START_ADDRESS, FLASH_MAX);

    data_flash[0]= peri_control & 0x0FF;
    data_flash[1]= ((u8) ((unsigned int) ((unsigned int) peri_control) &
0x0FF00)/((unsigned int) (256))));
    data_flash[2]= ((u8) ((unsigned int) ((unsigned int) peri_control) &
0x0FF0000)/((unsigned int) (65536))));
    data_flash[3]= ((u8) ((unsigned int) ((unsigned int) peri_control) &
0x0FF000000)/((unsigned int) (16777216))));
```

```
    data_flash[4]= code_size & 0x0FF;
    data_flash[5]= ((u8) ((unsigned int) ((((unsigned int) code_size) &
0x0FF00)/((unsigned int) (256)))));
    data_flash[6]= ((u8) ((unsigned int) ((((unsigned int) code_size) &
0x0FF0000)/((unsigned int) (65536)))));
    data_flash[7]= ((u8) ((unsigned int) ((((unsigned int) code_size) &
0x0FF000000)/((unsigned int) (16777216)))));
    data_flash[8]= tx_period & 0x0FF;
    data_flash[9]= ((u8) ((unsigned int) ((((unsigned int) tx_period) &
0x0FF00)/((unsigned int) (256)))));
    data_flash[10]= ((u8) ((unsigned int) ((((unsigned int) tx_period) &
0x0FF0000)/((unsigned int) (65536)))));
    data_flash[11]= ((u8) ((unsigned int) ((((unsigned int) tx_period) &
0x0FF000000)/((unsigned int) (16777216)))));

    write_size= PARAM_SIZE + NUM_BEACONS*BYTES_PER_SAMPLE*code_size;
    // write_size= PARAM_SIZE;

    // Number of pages to write
    no_page= (int) (write_size/PAGE_SIZE);
    if (no_page*PAGE_SIZE<write_size)
        no_page++;

    // Flash writing
    for (page_cnt = 0; page_cnt < no_page; page_cnt++)
    {
        for (i=0; i<PAGE_SIZE; i++)
            WriteBuffer[DATA_OFFSET+i]= data_flash[page_cnt*PAGE_SIZE+i];

        FlashWrite(&QspiInstance, (page_cnt*PAGE_SIZE) + START_ADDRESS,
                   PAGE_SIZE, WRITE_CMD);
    }

    // Status = XFlash_Write (&FlashInstance, START_ADDRESS, write_size,
data_flash);
}

void FlashRead(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command)
{
    // Setup the write command with the specified address and data for the
FLASH
    WriteBuffer[COMMAND_OFFSET]   = Command;
    WriteBuffer[ADDRESS_1_OFFSET] = (u8)((Address & 0xFF0000) >> 16);
    WriteBuffer[ADDRESS_2_OFFSET] = (u8)((Address & 0xFF00) >> 8);
    WriteBuffer[ADDRESS_3_OFFSET] = (u8)(Address & 0xFF);

    if ((Command == FAST_READ_CMD) || (Command == DUAL_READ_CMD) ||
        (Command == QUAD_READ_CMD))
    {
        ByteCount += DUMMY_SIZE;
    }

     /* Send the read command to the FLASH to read the specified number
     * of bytes from the FLASH, send the read command and address and
     * receive the specified number of bytes of data in the data buffer
     */
    XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, ReadBuffer,
                ByteCount + OVERHEAD_SIZE);
}

void FlashWrite(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command)
{
    u8 WriteEnableCmd = { WRITE_ENABLE_CMD };
```

```
    u8 ReadStatusCmd[] = { READ_STATUS_CMD, 0 };  /* must send 2 bytes */
    u8 FlashStatus[2];

    /*
     * Send the write enable command to the FLASH so that it can be
     * written to, this needs to be sent as a seperate transfer before
     * the write
     */
    XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
                sizeof(WriteEnableCmd));


    /*
     * Setup the write command with the specified address and data for the
     * FLASH
     */
    WriteBuffer[COMMAND_OFFSET]   = Command;
    WriteBuffer[ADDRESS_1_OFFSET] = (u8)((Address & 0xFF0000) >> 16);
    WriteBuffer[ADDRESS_2_OFFSET] = (u8)((Address & 0xFF00) >> 8);
    WriteBuffer[ADDRESS_3_OFFSET] = (u8)(Address & 0xFF);

    /*
     * Send the write command, address, and data to the FLASH to be
     * written, no receive buffer is specified since there is nothing to
     * receive
     */
    XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, NULL,
                ByteCount + OVERHEAD_SIZE);

    /*
     * Wait for the write command to the FLASH to be completed, it takes
     * some time for the data to be written
     */
    while (1) {
        /*
         * Poll the status register of the FLASH to determine when it
         * completes, by sending a read status command and receiving the
         * status byte
         */
        XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd, FlashStatus,
                    sizeof(ReadStatusCmd));

        /*
         * If the status indicates the write is done, then stop waiting,
         * if a value of 0xFF in the status byte is read from the
         * device and this loop never exits, the device slave select is
         * possibly incorrect such that the device status is not being
         * read
         */
        if ((FlashStatus[1] & 0x01) == 0) {
            break;
        }
    }
}

void FlashErase(XQspiPs *QspiPtr, u32 Address, u32 ByteCount)
{
    u8 WriteEnableCmd = { WRITE_ENABLE_CMD };
    u8 ReadStatusCmd[] = { READ_STATUS_CMD, 0 };  /* must send 2 bytes */
    u8 FlashStatus[2];
    u32 Sector;

    /*
```

```
     * If erase size is same as the total size of the flash, use bulk erase
     * command
     */
    if (ByteCount == (NUM_SECTORS * SECTOR_SIZE)) {
        /*
         * Send the write enable command to the FLASH so that it can be
         * written to, this needs to be sent as a seperate transfer
         * before the erase
         */
        XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
                  sizeof(WriteEnableCmd));

        /*
         * Setup the bulk erase command
         */
        WriteBuffer[COMMAND_OFFSET]   = BULK_ERASE_CMD;

        /*
         * Send the bulk erase command; no receive buffer is specified
         * since there is nothing to receive
         */
        XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, NULL,
                    BULK_ERASE_SIZE);

        /*
         * Wait for the erase command to the FLASH to be completed
         */
        while (1) {
            /*
             * Poll the status register of the device to determine
             * when it completes, by sending a read status command
             * and receiving the status byte
             */
            XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd,
                        FlashStatus,
                        sizeof(ReadStatusCmd));

            /*
             * If the status indicates the write is done, then stop
             * waiting; if a value of 0xFF in the status byte is
             * read from the device and this loop never exits, the
             * device slave select is possibly incorrect such that
             * the device status is not being read
             */
            if ((FlashStatus[1] & 0x01) == 0) {
                break;
            }
        }

        return;
    }

    /*
     * If the erase size is less than the total size of the flash, use
     * sector erase command
     */
    for (Sector = 0; Sector < ((ByteCount / SECTOR_SIZE) + 1); Sector++) {
        /*
         * Send the write enable command to the SEEPOM so that it can be
         * written to, this needs to be sent as a seperate transfer
         * before the write
         */
        XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
```

```
                    sizeof(WriteEnableCmd));

        /*
         * Setup the write command with the specified address and data
         * for the FLASH
         */
        WriteBuffer[COMMAND_OFFSET]   = SEC_ERASE_CMD;
        WriteBuffer[ADDRESS_1_OFFSET] = (u8)(Address >> 16);
        WriteBuffer[ADDRESS_2_OFFSET] = (u8)(Address >> 8);
        WriteBuffer[ADDRESS_3_OFFSET] = (u8)(Address & 0xFF);

        /*
         * Send the sector erase command and address; no receive buffer
         * is specified since there is nothing to receive
         */
        XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, NULL,
                    SEC_ERASE_SIZE);

        /*
         * Wait for the sector erse command to the FLASH to be completed
         */
        while (1) {
            /*
             * Poll the status register of the device to determine
             * when it completes, by sending a read status command
             * and receiving the status byte
             */
            XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd,
                        FlashStatus,
                        sizeof(ReadStatusCmd));

            /*
             * If the status indicates the write is done, then stop
             * waiting, if a value of 0xFF in the status byte is
             * read from the device and this loop never exits, the
             * device slave select is possibly incorrect such that
             * the device status is not being read
             */
            if ((FlashStatus[1] & 0x01) == 0) {
                break;
            }
        }

        Address += SECTOR_SIZE;
    }
}



/////////////////////////////////////////////////////////////////////////////
//////////
// Peripheral functions
/////////////////////////////////////////////////////////////////////////////
//////////

void writeRegPERI (Xuint32 peri_control, Xuint32 code_size, Xuint32
tx_period)
{
    per_pnt[CODESIZE_REG_ADDR] = (code_size & 0x0FFFFF);
    // per_pnt[TXPERIOD_REG_ADDR] = (tx_period & 0x0FFF);
    per_pnt[TXPERIOD_REG_ADDR] = (tx_period & 0x0FFFF);
    per_pnt[CONTROL_REG_ADDR] = (peri_control & 0x0FF);
}
```

```
void writeDataPERI (int addr_index, Xuint32* code_data)
{
    Xuint32 wr_data[NUM_BEACONS];


    wr_data[0]= (code_data[0] + (code_data[5]*(1<<16)));
    wr_data[1]= (code_data[1] + (code_data[6]*(1<<16)));
    wr_data[2]= (code_data[2] + (code_data[7]*(1<<16)));
    wr_data[3]= (code_data[3] + (code_data[8]*(1<<16)));
    wr_data[4]= (code_data[4] + (code_data[9]*(1<<16)));

    per_pnt[RAM1_ADDR/4 + addr_index] = wr_data[0];
    per_pnt[RAM2_ADDR/4 + addr_index] = wr_data[1];
    per_pnt[RAM3_ADDR/4 + addr_index] = wr_data[2];
    per_pnt[RAM4_ADDR/4 + addr_index] = wr_data[3];
    per_pnt[RAM5_ADDR/4 + addr_index] = wr_data[4];
}

void writeDDR2PERI (int per_index, int addr_index)
{
    /*if (addr_index>48508 && addr_index<48512)
        xil_printf ("\nIndex: %d;\t dato1: %d;\t dato2: %d.", addr_index,
b1_ddr_code[addr_index] & 0x0FFFF, (b1_ddr_code[addr_index] &
0x0FFF0000)/65536);
*/
    per_pnt[RAM1_ADDR/4 + per_index] = b1_ddr_code[addr_index];
    per_pnt[RAM2_ADDR/4 + per_index] = b2_ddr_code[addr_index];
    per_pnt[RAM3_ADDR/4 + per_index] = b3_ddr_code[addr_index];
    per_pnt[RAM4_ADDR/4 + per_index] = b4_ddr_code[addr_index];
    per_pnt[RAM5_ADDR/4 + per_index] = b5_ddr_code[addr_index];
}

void writeDDR2PERI_Null (int per_index)
{
    /*if (addr_index>48508 && addr_index<48512)
        xil_printf ("\nIndex: %d;\t dato1: %d;\t dato2: %d.", addr_index,
b1_ddr_code[addr_index] & 0x0FFFF, (b1_ddr_code[addr_index] &
0x0FFF0000)/65536);
*/
    per_pnt[RAM1_ADDR/4 + per_index] = PER_NULL_VALUE;
    per_pnt[RAM2_ADDR/4 + per_index] = PER_NULL_VALUE;
    per_pnt[RAM3_ADDR/4 + per_index] = PER_NULL_VALUE;
    per_pnt[RAM4_ADDR/4 + per_index] = PER_NULL_VALUE;
    per_pnt[RAM5_ADDR/4 + per_index] = PER_NULL_VALUE;
}

////////////////////////////////////////////////////////////////////////////
//////////
// DMA/DDR functions
////////////////////////////////////////////////////////////////////////////
//////////

void writeDataDDR (int addr_index, Xuint32* code_data)
{
    Xuint32 wr_data[NUM_BEACONS];


    wr_data[0]= (code_data[0] + (code_data[5]*(1<<16)));
    wr_data[1]= (code_data[1] + (code_data[6]*(1<<16)));
    wr_data[2]= (code_data[2] + (code_data[7]*(1<<16)));
    wr_data[3]= (code_data[3] + (code_data[8]*(1<<16)));
```

```
    wr_data[4]= (code_data[4] + (code_data[9]*(1<<16)));

    b1_ddr_code[addr_index] = wr_data[0];
    b2_ddr_code[addr_index] = wr_data[1];
    b3_ddr_code[addr_index] = wr_data[2];
    b4_ddr_code[addr_index] = wr_data[3];
    b5_ddr_code[addr_index] = wr_data[4];
}


//////////////////////////////////////////////////////////////////////////////
//////////
// BCU Interruption function
//////////////////////////////////////////////////////////////////////////////
//////////
void BCUISR (void *Callback)
{
    Xuint32 aux;
    int status, i;
    int bank_offset;


    // Peripheral status reading
    status= per_pnt[STATUS_REG_ADDR];
    //xil_printf ("Interruption received, status %d.\n", status);

    if (status & 0x02) // TX end
    {
        for (i=0; i<SIZE_PER_BEACON/2; i++)
        {
            writeDDR2PERI (i, i);
             /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t
B4 %X\t B5 %X.", i, i, b1_ddr_code[i],
                        b2_ddr_code[i], b3_ddr_code[i], b4_ddr_code[i],
b5_ddr_code[i]);*/
        }
        /* i--;
         xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t B4 %X\t
B5 %X.", i, i, b1_ddr_code[i],
                  b2_ddr_code[i], b3_ddr_code[i], b4_ddr_code[i],
b5_ddr_code[i]); */

        // rt_code_cnt= SIZE_PER_BEACON/4;
        rt_code_cnt= SIZE_PER_BEACON/2;

        int_count= 1;
        // xil_printf("\n No. int: %d.", int_count);
    }
    else
    {
        if (status & 0x01)
            bank_offset= 0;
        else
            bank_offset= SIZE_PER_BEACON/4;

        /* if (status & 0x01)
            bank_offset= SIZE_PER_BEACON/4;
        else
            bank_offset= 0; */

        /* if (int_count & 0x01)
            bank_offset= SIZE_PER_BEACON/4;
        else
```

```
            bank_offset= 0;*/

        if (rt_code_cnt>=code_size/2)
        {
            for (i=0; i<SIZE_PER_BEACON/4; i++)
            {
                writeDDR2PERI_Null (bank_offset+i);
                // xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3
%X\t B4 %X\t B5 %X.", i, bank_offset+i, 0,0,0,0,0);
            }
            // i--;
            // xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t B4
%X\t B5 %X.", i, bank_offset+i, 0,0,0,0,0);
        }
        else if ((rt_code_cnt + SIZE_PER_BEACON/4>code_size/2) &&
(rt_code_cnt< code_size/2))
        {
            for (i=0; i<code_size/2-rt_code_cnt; i++)
            {
                writeDDR2PERI (bank_offset+i, rt_code_cnt+i);
                /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3
%X\t B4 %X\t B5 %X.", rt_code_cnt+i, bank_offset+i,
b1_ddr_code[rt_code_cnt+i],
                    b2_ddr_code[rt_code_cnt+i], b3_ddr_code[rt_code_cnt+i],
b4_ddr_code[rt_code_cnt+i], b5_ddr_code[rt_code_cnt+i]); */
            }
            // i--;
            /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t B4
%X\t B5 %X.", rt_code_cnt+i, bank_offset+i, b1_ddr_code[rt_code_cnt+i],
                            b2_ddr_code[rt_code_cnt+i],
b3_ddr_code[rt_code_cnt+i], b4_ddr_code[rt_code_cnt+i],
b5_ddr_code[rt_code_cnt+i]);*/

            for (i=code_size/2-rt_code_cnt; i<SIZE_PER_BEACON/4; i++)
            {
                writeDDR2PERI_Null (bank_offset+i);
                // xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3
%X\t B4 %X\t B5 %X.", rt_code_cnt+i, bank_offset+i, 0,0,0,0,0);
            }
            // i--;
            /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t B4
%X\t B5 %X.", rt_code_cnt+i, bank_offset+i, 0,0,0,0,0);*/

            rt_code_cnt= code_size/2;

        }
        else
        {
            for (i=0; i<SIZE_PER_BEACON/4; i++)
            {
                writeDDR2PERI (bank_offset+i, rt_code_cnt+i);
                /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3
%X\t B4 %X\t B5 %X.", rt_code_cnt+i, bank_offset+i,
b1_ddr_code[rt_code_cnt+i],
                        b2_ddr_code[rt_code_cnt+i],
b3_ddr_code[rt_code_cnt+i], b4_ddr_code[rt_code_cnt+i],
b5_ddr_code[rt_code_cnt+i]); */
            }

            // i--;
            /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t B4
%X\t B5 %X.", rt_code_cnt+i, bank_offset+i, b1_ddr_code[rt_code_cnt+i],
```

```
                    b2_ddr_code[rt_code_cnt+i], b3_ddr_code[rt_code_cnt+i],
b4_ddr_code[rt_code_cnt+i], b5_ddr_code[rt_code_cnt+i]); */


            rt_code_cnt= rt_code_cnt + SIZE_PER_BEACON/4;
        }

        int_count++;
        // xil_printf("\n No. int: %d.", int_count);

        // Peripheral status reading
        status= per_pnt[STATUS_REG_ADDR];

        if (status & 0x02) // TX end
        {
            for (i=0; i<SIZE_PER_BEACON/2; i++)
            {
                writeDDR2PERI (i, i);
                 /* xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3
%X\t B4 %X\t B5 %X.", i, i, b1_ddr_code[i],
                        b2_ddr_code[i], b3_ddr_code[i], b4_ddr_code[i],
b5_ddr_code[i]);*/
            }
            /* i--;
             xil_printf ("\nData %d\t Dest: %d\t B1 %X\t B2 %X\t B3 %X\t B4
%X\t B5 %X.", i, i, b1_ddr_code[i],
                    b2_ddr_code[i], b3_ddr_code[i], b4_ddr_code[i],
b5_ddr_code[i]); */

            // rt_code_cnt= SIZE_PER_BEACON/4;
            rt_code_cnt= SIZE_PER_BEACON/2;

            int_count= 1;
            // xil_printf("\n No. int: %d.", int_count);
        }
    }

    // Peripheral status reading
    /* status= per_pnt[STATUS_REG_ADDR];

    if (status & 0x02) // TX end
    {
        for (i=0; i<SIZE_PER_BEACON/2; i++)
            writeDDR2PERI (i, i);

        rt_code_cnt= SIZE_PER_BEACON/2;
    } */

    // Interruption reset
    aux= per_pnt[CONTROL_REG_ADDR];
    per_pnt[CONTROL_REG_ADDR]= aux + 0x08;
    per_pnt[CONTROL_REG_ADDR]= aux;
}



////////////////////////////////////////////////////////////////////////////
//////////
// WIFI functions
////////////////////////////////////////////////////////////////////////////
//////////

void PmodWIFI_Initialize()
```

```
{
   setPmodWifiAddresses( XPAR_PMODWIFI_0_AXI_LITE_SPI_BASEADDR,
                         XPAR_PMODWIFI_0_AXI_LITE_WFGPIO_BASEADDR,
                         XPAR_PMODWIFI_0_AXI_LITE_WFCS_BASEADDR,
                         XPAR_PMODWIFI_0_S_AXI_TIMER_BASEADDR );

   setPmodWifiIntVector(PMODWIFI_VEC_ID);
}

void print_app_header ()
{
    xil_printf ("\n\r\n\r------- TCP SERVER ------\n\r");
}


void print_ip_settings (IPv4 *ip, IPv4 *mask, IPv4 *gw)
{
    print_ip ("Board IP: ", ip);
    print_ip ("Netmask : ", mask);
    print_ip ("Gateway : ", gw);
}


void print_ip (char *msg, IPv4 *ip)
{
    print (msg);
    xil_printf ("%d.%d.%d.%d\n\r", ip->u8[0], ip->u8[1], ip->u8[2], ip-
>u8[3]);
}

void StartWIFI_Aplication (void)
{
    /*WIFI CONECTION VARIABLES*/
    u8 rgbRead[1024];
    int cbRead = 0;
    int count = 0;
    unsigned tStart = 0;
    unsigned tWait = 5000;
    unsigned int timeout = 0;


    estado_conexion = CONNECT;

    while (1)
    {
        switch (estado_conexion)
        {
        case CONNECT:
         if ( WiFiConnectMacro() )
         {

            xil_printf("Connection Created\r\n");

            deIPcK.begin(ipMyStatic);

            /* Restablecer la interrupci□CUI, ya que al inicalizarse el m□o
WIFI,
             * este deshabilita todas las interrupciones*/
            RestartIntrSystem();

            estado_conexion = LISTEN;
         }
         else if ( IsIPStatusAnError(status) )
```

```
         {
            xil_printf("Unable to make connection, status: 0x%X\r\n",
status);

            /* Restablecer la interrupci□CUI, ya que al inicalizarse el m□o
WIFI,
             * este deshabilita todas las interrupciones*/
            RestartIntrSystem();

            estado_conexion = EXIT;
         }

         break;

        // Crear un servidor que escuche en un determinado puerto
        case LISTEN:
         if ( deIPcK.tcpStartListening(portServer, tcpServer) )
         {
            for (int i = 0; i < cTcpClients; i++)
            {
               tcpServer.addSocket( rgTcpClient[i] );
            }
         }

         estado_conexion = ISLISTENING;

         break;

        case ISLISTENING:

         count = tcpServer.isListening();

         if (count > 0)
         {
            deIPcK.getMyIP( ipMyStatic );
            xil_printf("Server started on %d.%d.%d.%d:%d\r\n",
ipMyStatic.u8[0],

ipMyStatic.u8[1],

ipMyStatic.u8[2],

ipMyStatic.u8[3],
                                                          portServer );

            xil_printf("%d sockets listening on port: %d\r\n",
                        count,
                        portServer);

            estado_conexion = AVAILABLECLIENT;

         }
         else
         {
             estado_conexion = WAITISLISTENING;
         }
         break;

        case WAITISLISTENING:
         if ( tcpServer.isListening() > 0 )
         {
             estado_conexion = ISLISTENING;
         }
```

```
         break;

        // Esperar a un cliente que realice una petici□e conexi□        case
AVAILABLECLIENT:
         if ( ( count = tcpServer.availableClients() ) > 0 )
         {
            xil_printf("Got %d clients pending\r\n", count);
            estado_conexion = ACCEPTCLIENT;
         }
         break;

        // Aceptar la conexi□on el cliente
        case ACCEPTCLIENT:

         // Accept the client
         if ( (ptcpClient = tcpServer.acceptClient() ) != NULL  &&
               ptcpClient->isConnected() )
         {
            xil_printf("Got a Connection\r\n");
            estado_conexion = READ;
            tStart = (unsigned) SYSGetMilliSecond();
         }

         // This probably won't happen unless the connection is dropped
         // if it is, just release our socket and go back to listening
         else
         {
             estado_conexion = CLOSE;
         }
         break;

        // Wait for the read, but if too much time elapses (5 seconds)
        // we will just close the tcpClient and go back to listening
        case READ:

        // See if we got anything to read
        if ( (cbRead = ptcpClient->available()) > 0 )
        {
            timeout = 0;

            cbRead = cbRead < (int) sizeof(rgbRead) ? cbRead :
sizeof(rgbRead);

            // indicate that the packet has been received
            cbRead = ptcpClient->readStream(rgbRead, cbRead);
            xil_printf("Got %d bytes\r\n", cbRead);

            if ( newConfiguration((char *) rgbRead, cbRead) == 1 )
            {
                estado_conexion = WRITE;
            }

        }

         // If connection was closed by the user
         else if ( !ptcpClient->isConnected() )
         {
             estado_conexion = CLOSE;
         }

        /*Si no se lee nada se lleva a cabo un timeout*/
         else
         {
```

```
            timeout++;

            if (timeout == 1e7)
            {
                estado_conexion = CLOSE;
                timeout = 0;
            }
        }


        break;

       // Se realiza un envdel ACK
       case WRITE:
        if (ptcpClient->isConnected())
        {
           xil_printf("Sending ACK... \n");


           if ( sendACK (2*code_cnt) == -1 )
           {
               xil_printf ("\nError sending ACK\n");
           }

           estado_conexion = CLOSE;

        }

        // The connection was closed on us, go back to listening
        else
        {
           xil_printf("Unable to write back.\r\n");
           estado_conexion = CLOSE;
        }
        break;

       // Close our tcpClient and go back to listening
       case CLOSE:
        xil_printf("Closing connection\r\n");

        if (ptcpClient)
        {
           ptcpClient->close();
        }

        tcpServer.addSocket(*ptcpClient);

        xil_printf("\r\n");
        estado_conexion = ISLISTENING;
        break;

       // Something bad happened, just exit out of the program
       case EXIT:

        tcpServer.close();
        xil_printf("Something went wrong, sketch is done.\r\n");
        estado_conexion = DONE;
        break;

       // Do nothing in the loop
       case DONE:

       break;
```

```
        default:
         break;
        }

        // Every pass through loop(), keep the stack alive
        DEIPcK::periodicTasks();
    }
}



bool newConfiguration (char *check, int long_cad)
{
    int     num, mss_counter;
    int     i;
    bool    end = 0;

    for (mss_counter = 0; mss_counter < long_cad; mss_counter++)
    {
        num = ( *(check + mss_counter) ) - 48;
        if ( ( num > -1 ) &&  ( num < 10 ) )
        {
            integer = 1;
        }
        else
        {
            integer = 0;
        }

        switch (state)
        {
            case qWait: // Communication init with key n/N
                if ((*(check+mss_counter)=='n')||(*(check+mss_counter)=='N'))
                {
                    code_size= 0;
                    tx_period= 0;
                    code_cnt= 0;
                    rt_code_cnt= 0;
                    int_count= 0;

                    // idxFlash= PARAM_SIZE + DATA_OFFSET;
                    idxFlash= PARAM_SIZE;
                    for (i=0; i<2*NUM_BEACONS; i++)
                        code_data[i]= 0;

                    beacon= 0;

                    xil_printf ("\nNew Configuration\n");
                    state = qMasterSlave;
                    writeRegPERI (0, code_size, tx_period); // Disable system
                }
                else if
((*(check+mss_counter)=='r')||(*(check+mss_counter)=='R'))
                {
                    // readAndSendConf (tpcb);
                    xil_printf ("\nFunction no supported.\n\r");
                }
                break;

            case qMasterSlave:  //LPS master or slave [1/0]
                if ((*(check+mss_counter)=='c')||(*(check+mss_counter)=='C'))
```

```
                {
                    xil_printf ("\nConfiguration canceled at Master/Slave
mode\n");
                    state= qWait;
                }
                else if (integer==1)
                {
                    peri_control= num;
                }
                else if(*(check+mss_counter)==enter)
                {
                    state=qCodeSize;
                }
                break;

            case qCodeSize: // Code length
                if ((*(check+mss_counter)=='c')||(*(check+mss_counter)=='C'))
                {
                    xil_printf ("\nConfiguration canceled at code size
load\n");
                    state= qWait;
                }
                else if (integer==1)
                {
                    code_size= num + code_size*10;
                }
                else if (*(check+mss_counter)==enter)
                {
                    if (code_size>SIZE_PER_BEACON)  // limit 20k
                        ext_mode= 1;
                    else
                        ext_mode= 0;

                    state= qTxPeriod;
                }
                break;

            case qTxPeriod:
                if ((*(check+mss_counter)=='c')||(*(check+mss_counter)=='C'))
                {
                    xil_printf ("\nConfiguration canceled at TX Period
load\n");
                    state= qWait;
                }
                else if (integer==1)
                {
                    tx_period= num + tx_period*10;
                }
                else if (*(check+mss_counter)==enter)
                {
                    state= qCodes;

                    xil_printf ("\nControl: %d;\t Code size: %d;\t TX Period:
%d.\n", peri_control, code_size, tx_period);
                    writeRegPERI ((peri_control&0xFFFFFFFE),  code_size,
tx_period); // Not enable
                }
                break;

            case qCodes:    // Codes
                if ( (*(check+mss_counter) == 'c' ) || ( *(check+mss_counter)
== 'C' ) )
                {
```

```
                    xil_printf ("\nConfiguration canceled at codes load\n");
                    xil_printf ("Code byte index: %d\n", code_cnt);
                    state= qWait;
                }
                else if (integer==1)
                {
                    code_data[beacon]= num + code_data[beacon]*10;
                }
                else if (*(check+mss_counter)==enter)
                {
                    beacon++;
                    if (beacon== 2*NUM_BEACONS)
                    {
                        // xil_printf ("\nNoA. %d \t %d \t %d", code_cnt,
code_data[0], code_data[5]);
                        /* if (code_cnt>48508 && code_cnt<48512)
                        {
                            xil_printf ("\nCFG Index: %d;\t dato1: %d;\t
dato2: %d.", code_cnt, code_data[0], code_data[5]);
                        }*/
                        if (code_data[0]>code_data[5])
                        {
                            //xil_printf ("\nCFG Index: %d;\t dato1: %d;\t
dato2: %d.", code_cnt, code_data[0], code_data[5]);
                        }

                        if (ext_mode==0)
                            writeDataPERI(code_cnt, code_data);
                        else
                        {
                            writeDataDDR (code_cnt, code_data);
                            if (code_cnt<SIZE_PER_BEACON/2)
                            {
                                //xil_printf ("\nNoB. %d \t %d \t %d",
code_cnt, code_data[0], code_data[5]);
                                writeDataPERI(code_cnt, code_data);
                            }
                            else
                            {
                                rt_code_cnt= SIZE_PER_BEACON/2;
                                int_count= 0;
                            }
                        }
                        // xil_printf ("\nNo. %d", code_cnt);

                        code_cnt++;

                        for (i=0; i<2*NUM_BEACONS; i++)
                        {
                            // data_flash[idxFlash]= (code_data[i] & 0xff);
                            // data_flash[idxFlash+1]= ((code_data[i] &
0xff00)>>8);
                            data_flash[idxFlash]= (code_data[i] & 0x0ff);
                            data_flash[idxFlash+1]= ((code_data[i] &
0x0ff00)>>8);
                            idxFlash+= 2;
                        }

                        if (code_cnt==code_size/2)
                        {
                            xil_printf ("\nNo. final %d\n", code_cnt);
                            state= qWait;
                            xil_printf ("ACK\n");
```

```
                                xil_printf ("ACK %d %d\n\n", idxFlash, code_cnt);
                                writeFlash (peri_control, code_size, tx_period);
                                writeRegPERI (peri_control, code_size,
tx_period); // Enable system

                                /*if (sendACK (2*code_cnt) == -1)
                                {
                                    xil_printf ("\nError sending ACK\n");
                                }*/

                                /* for (i=102300; i<102600; i++)
                                {
                                    xil_printf ("\nNo. %d\t Data %x", i,
data_flash[i]);
                                } */

                                end = 1;
                            }

                            for (i=0; i<2*NUM_BEACONS; i++)
                                code_data[i]= 0;
                            beacon= 0;
                        }
                    }
                    break;

                default:
                    break;
            }
        }

        return end;
}



int sendACK (int data)
{
     if (ptcpClient->isConnected())
     {
        //xil_printf("Writing... \r\n");

        ptcpClient->writeStream((u8 *)&data, sizeof (int));
        /*estado_conexion = READ;*/
        /*tStart = (unsigned) SYSGetMilliSecond();*/
        return 0;
     }

     // The connection was closed on us, go back to listening
     else
     {
        xil_printf("Unable to write back.\r\n");
        estado_conexion = CLOSE;
        return -1;
     }

}
```

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR

Universidad de Alcalá